# Jugadores Automáticos basados en Deep Reinforcement Learning


**Trabajo Fin de Grado**

**Rodrigo Bravo Antón**
**Javier García Rodriguez**
**David Alba Corral**

**Facultad de Informática**
**Universidad Complutense de Madrid**

**Septiembre 2018**

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

# Jugadores Automáticos basados en Deep Reinforcement Learning

*Trabajo Fin de Grado*
**Grado en Ingenieria Informática**

*Dirigido por los Doctores*
Antonio Alejandro Sánchez Ruiz-Granados
Pedro Pablo Gómez Martín

**Facultad de Informática**
**Universidad Complutense de Madrid**

**Septiembre 2018**

*Al gran Cuevas,*
*implacable en el BANG!,*
*afable en la amistad*

# Agradecimientos

*A nuestros padres, compañeros y amigos que nos han apoyado y acompañado durante esta etapa universitaria que con este trabajo llega a su fin. También a los docentes que nos hemos encontrado durante la misma, especialmente a nuestros tutores Antonio y Pedro Pablo por instruirnos y aguantarnos a lo largo de este camino y ayudarnos a conseguir que llegara a buen puerto.*

*Muchas gracias a todos.*

# Resumen del proyecto

En los últimos años, el Deep Reinforcement Learning se ha convertido en una de las ramas más prometedoras del área de la inteligencia artificial. En este proyecto vamos a estudiar dicha rama y todos los componentes que la forman. Posteriormente, para aplicar los conocimientos adquiridos, programaremos un problema sencillo en el que un agente tiene que encontrar la salida en un mapa a modo de toma de contacto, y, a continuación, pondremos a prueba la corrección de la implementación de nuestro agente en un sistema de pruebas conocido internacionalmente como es OpenAI Gym.

Todo el software desarrollado para este trabajo está disponible en GitHub en la URL:

`https://github.com/robrav01/TFG-RLDL`

## Palabras clave

Deep Reinforcement Learning

Aprendizaje por refuerzo

Q-Learning

Redes neuronales

Aprendizaje profundo

DeepMind

OpenAI

Conductismo

# Project's Summary

In the last few years, Deep Reinforcement Learning has become one of the most promising subjects in the Artificial Intelligence area. In this project, we are going to study this subject and all the parts it is composed by. Later, for applying the acquired knowledge, we will program a simple problem in which an agent has to find the exit in a map as a starting point, and, then, we will prove the correctness of the implementation of our agent in an internationally known test system as OpenAI Gym.

All the software developed for this project is available on GitHub at the URL:

`https://github.com/robrav01/TFG-RLDL`

## Keywords

Deep Reinforcement Learning

Reinforcement Learning

Q-Learning

Neural Networks

Deep Learning

DeepMind

OpenAI

Behaviorism

# Acrónimos

**A3C** - Asynchronous Advantage Actor-Critic Agent

**API** - Application Programming Interface

**CPU** - Central Processing Unit

**DL** - Deep Learning

**DQN** - Deep Q-Network

**DRL** - Deep Reinforcement Learning

**DRQN** - Deep Recurrent Q-Networks

**GPU** - Graphic Processing Unit

**IA** - Inteligencia Artificial

**MDP** - Markov Decision Process

**MOBA** - Multiplayer Online Battle Arena

**MSE** - Mean Square Error

**ReLU** - Rectified Linear Unit

**RL** - Reinforcement Learning

**UNREAL** - UNsupervised REinforcement and Auxiliary Learning

# Índice

**Agradecimientos** vii

**Resumen del proyecto** ix

**Acrónimos** xiii

**1. Introducción** 1
- 1.1. Objetivos . . . 1
- 1.2. Plan de trabajo . . . 2
- 1.3. Estructura de la memoria . . . 2

**2. Antecedentes** 7
- 2.1. Introducción . . . 7
- 2.2. Reinforcement Learning. Q-Learning . . . 8
  - 2.2.1. ¿Qué es el Reinforcement Learning? . . . 10
  - 2.2.2. Elementos del Reinforcement Learning . . . 10
    - Estados . . . 10
    - Acciones . . . 10
    - Recompensas . . . 11
  - 2.2.3. *Markov Decision Process* . . . 12
    - Future return . . . 14
    - Value learning . . . 15
  - 2.2.4. Proceso de aprendizaje . . . 16
    - El dilema exploración-explotación . . . 17
  - 2.2.5. Q-Learning . . . 18
    - La ecuación de Bellman . . . 19
- 2.3. Deep Learning. Redes Neuronales . . . 20
  - 2.3.1. Estructura de las redes neuronales . . . 21
  - 2.3.2. Definición del modelo . . . 23
    - Parámetros . . . 23
    - Regresión lineal . . . 24
    - Entrenamiento y reducción de la pérdida de una neurona 25

2.3.2.1. Backpropagation . . . . . 28
Backpropagation . . . . . 28
2.3.3. Grandes proyectos basados en Deep Learning . . . . . 29
Google Vision . . . . . 29
Amazon Rekognition . . . . . 30
2.4. Deep Reinforcement Learning . . . . . 30
2.4.1. Deep Q-Networks . . . . . 31
2.4.2. Estabilidad del aprendizaje. Problemas y soluciones . . 32
Target Q-Network . . . . . 33
Experience Replay . . . . . 33
2.4.3. Mejoras de las Deep Q-Networks . . . . . 34
Deep Recurrent Q-Networks (DRQN) . . . . . 34
Asynchronous Advantage Actor-Critic Agent (A3C) . 34
UNsupervised REinforcement and Auxiliary Learning 35
2.4.4. Costes. Tiempo y recursos hardware . . . . . 35
2.4.5. Otros proyectos destacados . . . . . 37
AlphaGo . . . . . 37
OpenAI Dota 2 . . . . . 39

**3. Descripción del problema y desarrollo 45**
3.1. Lenguajes de programación y herramientas utilizadas . . . . . 45
3.2. Descripción del problema . . . . . 47
3.3. Solucion Q-Learning . . . . . 48
Resultados . . . . . 49
Limitaciones y conclusiones . . . . . 50
3.4. Solución Deep Reinforcement Learning . . . . . 51
Resultados . . . . . 56
Limitaciones y conclusiones . . . . . 57

**4. OpenAI 59**
4.1. CartPole . . . . . 60
Modificaciones realizadas . . . . . 60
Topología de la red e hiperparámetros . . . . . 61
Resultados obtenidos . . . . . 61
4.2. Taxi . . . . . 62
Modificaciones realizadas . . . . . 63
Topología de la red e hiperparámetros . . . . . 64
Resultados obtenidos . . . . . 64

**5. Conclusiones 67**
5.1. Trabajo Futuro . . . . . 68

**6. Aportación de los participantes** **73**
6.1. David Alba Corral . . . . . 74
6.2. Rodrigo Bravo Antón . . . . . 75
6.3. Javier García Rodriguez . . . . . 76

**Bibliografía** **79**

# Índice de figuras


2.1. Campos de investigación relacionados con el aprendizaje por refuerzo. [38] 9
2.2. Visión general de un MDP. [12] . . . . . 12
2.3. Nodo/Neurona. [11] . . . . . 22
2.4. Capas de una red neuronal. [11] . . . . . 22
2.5. Ejemplo de uso de las capas para la partición del problema. [4] . . . . . 22
2.6. Chirridos por minuto vs. Temperatura en grados Celsius. [18] . . . . . 25
2.7. Relación lineal. [18] . . . . . 25
2.8. Pérdida alta en el primer modelo; Pérdida baja en el segundo modelo. [18] 26
2.9. Elección de un punto de partida. [18] . . . . . 27
2.10. Descenso de gradiente. [18] . . . . . 27
2.11. Siguiente valor de $w_i$. [18] . . . . . 27
2.12. Superficie para un error cuadrático. [12] . . . . . 28
2.13. Captura del juego Space Invaders de la consola Atari. [31] . . . . . 31
2.14. Captura del juego *Breakout* de la consola Atari. [31] . . . . . 32
2.15. El agente trata de aprender a subir unas escaleras sin caerse. [21] . . . 36
2.16. Fotografía de una partida de go. [15] . . . . . 37
2.17. Imagen del documental AlphaGo [8] realizado por Netflix en el que se muestra el enfrentamiento entre AlphaGo y Lee Sedol. [6] . . . . . 39
2.18. Captura del videojuego Dota 2. [3] . . . . . 40

3.1. Logo de la herramienta Keras. [17] . . . . . 46
3.2. Logo de la librería Numpy. [40] . . . . . 46
3.3. Representación gráfica del problema. El círculo amarillo representa al agente; los rectángulos rojo y azul la posición inicial y la salida respectivamente. . . . . 48
3.4. Movimientos que realiza en cada episodio el agente. En azul los movimientos que tarda en total en completar el recorrido y en amarillo los movimientos que ha tardado en coger la moneda. . . . . 50

3.5. Los valores indicados en cada casilla representan la mejor acción posible utilizando sus iniciales en inglés (u-up, d-down, l-left, r-right). El valor que se encuentra más arriba indica la mejor acción si aún no se ha recogido la moneda y el que se encuentra más abajo indica la mejor acción una vez se ha recogido. . . . . 54
3.6. Se observa como incluso complicando el problema (ahora tiene que bajar a por la moneda y subir hacia la salida) es capaz de obtener la máxima puntuación. Además, se aprecia cómo incluir la moneda en el estado ayuda a su resolución. . . . . 56
3.7. Media de movimientos que tarda en el agente alcanzar la salida (en azul) y obtener la moneda (en amarillo) en cada episodio (300 movimientos). Los puntos con valores negativos son episodios en los que no se ha llegado a alcanzar la salida. . . . . 57

4.1. Captura de nuestro proyecto ejecutando el problema CartPole. [29] . . . 60
4.2. Media de movimientos tomados por el agente antes de que caiga el poste (eje y) en cada episodio de aprendizaje (eje x). . . . . 62
4.3. Captura de nuestro proyecto ejecutando el problema Taxi [30] de OpenAI. 62
4.4. Recompensas total (en gris) y promedio (en azul) obtenidas por el agente con experience replay en cada episodio (1000 pasos). Se estabiliza cerca del óptimo a partir de los 3000 episodios. . . . . 65
4.5. Recompensas total (en gris) y promedio (en azul) obtenidas por el agente utilizando para el entrenamiento todas las experiencias almacenadas en cada episodio. Se acerca al óptimo por primera vez en el episodio 2800. . 65

# Capítulo 1

# Introducción

El Deep Reinforcement Learning es un área del aprendizaje automático que consiste en una combinación de Reinforcement Learning (aprendizaje por refuerzo) con Deep Learning (aprendizaje profundo) y que se encuentra en auge en los últimos años. Tiene su origen en el año 2013, cuando el equipo de DeepMind sorprendió al mundo con un proyecto de inteligencia artificial en el que desarrolló un agente capaz de jugar a 7 juegos de la consola Atari utilizando como entrada únicamente los píxeles de la pantalla.

Desde entonces se han llevado a cabo grandes proyectos basados en esta tecnología, como por ejemplo AlphaGo, un agente de inteligencia artificial capaz de jugar al juego de mesa go, o OpenAI Dota 2, un jugador automático de este videojuego capaz de enfrentarse a jugadores profesionales y que ha cosechado tan buenos resultados que actualmente se está desarrollando una versión posterior con la habilidad de controlar al equipo completo.

En este proyecto hemos investigado y estudiado el área del Deep Reinforcement Learning con el fin de sintetizar sus puntos básicos y llevarlos a la práctica, tanto en un sencillo problema diseñado por nosotros mismos en el que un agente deberá recorrer un mapa rectangular con el fin de encontrar su salida y pudiendo recoger una moneda por el camino para aumentar su puntuación como en varios problemas propios de la herramienta OpenAI Gym, un conjunto de entornos de entrenamiento que permite poner a prueba algoritmos de aprendizaje.

## 1.1. Objetivos

1. Estudiar el campo del Reinforcement Learning, en concreto del Q-Learning, con el fin de comprender sus características y ser capaces de explicarlas de manera sencilla.

2. Aplicar los conocimientos obtenidos sobre Q-Learning para desarrollar un proyecto práctico y observar su funcionamiento por nosotros mismos.

3. Comprender el Deep Learning y obtener los conocimientos necesarios para utilizar correctamente las redes neuronales.

4. Estudiar en profundidad el campo del Deep Reinforcement Learning, desde su concepto más básico hasta las versiones más avanzadas, con el objetivo de conocer los métodos utilizados a la hora de desarrollar esta tecnología. Asimismo, ser capaces de explicar las bases que lo componen de manera que este proyecto pueda servir como punto de partida para aquellos interesados en esta materia.

5. Aplicar los conocimientos conjuntos de Q-Learning, Deep Learning y Deep Reinforcement Learning para resolver varios problemas de pequeña escala y comprobar sus cualidades y limitaciones.

## 1.2. Plan de trabajo

1. Estudio e investigación del Q-Learning y su posterior aplicación en un proyecto práctico con el fin de comprobar por nosotros mismos sus capacidades y limitaciones.

2. Estudio e investigación de las redes neuronales y el Deep Learning para adquirir las capacidades necesarias para su correcta utilización, de manera que podamos aplicarlas de manera conjunta con los conocimientos adquiridos sobre el Q-Learning.

3. Estudio e investigación del Deep Reinforcement Learning y su posterior aplicación en un proyecto práctico mediante el uso de todo lo aprendido anteriormente.

4. Empleo del agente desarrollado en el punto anterior en combinación con la herramienta OpenAI Gym con el fin de resolver varios problemas prácticos utilizando esta tecnología.

## 1.3. Estructura de la memoria

- **Capítulo 2**, **Antecedentes**. En este capítulo explicaremos todo el marco teórico necesario para entender el Deep Reinforcement Learning. Empezaremos por el Reinforcement Learning, seguiremos con el Deep Learning y terminaremos con el propio Deep Reinforcement Learning.

- **Capítulo 3**, **Descripción del problema y desarrollo**. En este capítulo aplicaremos los conocimientos teóricos obtenidos para resolver mediante ellos un sencillo problema diseñado por nosotros.
- **Capítulo 4**, **OpenAI**. En este capítulo utilizaremos los problemas ofrecidos por la herramienta OpenAI Gym para comprobar la corrección de la implementación del agente desarrollado en el capítulo anterior.
- **Capítulo 5**, **Conclusiones**. En este capítulo presentaremos las conclusiones que hemos obtenido durante el desarrollo de este proyecto.
- **Capítulo 6**, **Aportación de los participantes**. En este capítulo empezaremos presentado el trabajo realizado conjuntamente y luego el de cada uno de los integrantes.

# Introduction

Deep Reinforcement Learning is a Machine Learning subject which combines Reinforcement Learning and Deep Learning and it is booming in the last few years. It was born in 2013, when the developers of DeepMind astonished the world with an Artificial Intelligence project able to play seven Atari games using as input the pixels shown in the screen.

Since then, great projects of this technology has been developed. Some examples could be AlphaGo, an Artificial Intelligence agent able to play the game Go, or Open AI Dota 2, a bot that can play Dota 2 against a professional player achieving so unbelievable results that nowadays, there is another version in progress to play the five members of a team.

In this project, we have investigated and studied the area of Deep Reinforcement Learning in order to synthesize its main concepts and put them into practice, both in a simple problem designed by ourselves in which an agent must travel a rectangular map in order to find the exit and with the option of picking up a coin to increase its score as in several problems of the OpenAI toolkit, a set of training environments that allows to test learning algorithms.

## Objectives

1. Study Reinforcement Learning, more specifically Q-Learning, in order to comprehend its characteristics and be able to explain them in a simple way.
2. Apply the obtained knowledge about Q-Learning to develop a practical project and to analyze its working by ourselves.
3. Understand Deep Learning and obtain the necessary skills to use Neural Networks correctly.
4. Study Deep Reinforcement Learning deeply, from the most basic concept to the most advanced versions, with the purpose of knowing the methods for developing this technology. Also, be able to

explain the basics that compose it so this project it can be used as a starting point for those who are interested in this subject.

5. Apply our Q-Learning, Deep Learning and Deep Reinforcement Learning knowledge together to solve several small problems and prove its qualities and limitations.

## Work schedule

1. Study and research of Q-Learning for applying it in a practical project in order to check by ourselves its capabilities and limitations.
2. Study and research of Neural Networks and Deep Learning in order to obtain the necessary skills for using them correctly, so we can apply them together with our Q-Learning knowledge.
3. Study and research of Deep Reinforcement Learning for applying it in a practical project by using everything learned previously.
4. Use the bot developed in the previous point combined with the OpenAI Gym toolkit with the purpose of solving some practical problems using this technology.

## Structure of the project report

- **Chapter 2**, **Background**. In this chapter we will explain the whole theoretical framework needed to understand Deep Reinforcement Learning. We will begin with Reinforcement Learning, then Deep Learning and we will en with Deep Reinforcement Learning itself.
- **Chapter 3**, **Description of the problem and development**. In this chapter we will apply the theoretical knowledge obtained to solve through it a simple problem designed by ourselves.
- **Chapter 4**, **OpenAI**. In this chapter we will use the problems offered by the OpenAI Gym toolkit to check the correctness of the agent implementation developed in the previous chapter.
- **Chapter 5**, **Conclusions**. In this chapter we will present the conclusions that we have obtained during the development of this project.
- **Chapter 6**, **Contribution of the participants**. In this chapter we will start showing the work done jointly and then the one done by each participant.

# Capítulo 2

# Antecedentes

## 2.1. Introducción

La **inteligencia artificial** es una de las ramas de la computación que tiene como fin desarrollar programas capaces de realizar acciones propias de la inteligencia humana, como el aprendizaje, el razonamiento o la resolución de problemas.

Dentro del campo del aprendizaje encontramos el **aprendizaje automático** o *machine learning*, el cual busca desarrollar algoritmos con la capacidad de generalizar comportamientos y aprender patrones de actuación a partir de ejemplos concretos, de modo que sean capaces de resolver problemas sin haber sido programados explícitamente para ello.

De las diferentes técnicas de aprendizaje que forman parte del aprendizaje automático destacan las siguientes:

- **Aprendizaje supervisado** o *supervised learning*: es una técnica de aprendizaje cuyo método parte de una serie de ejemplos llamada conjunto de entrenamiento. Los ejemplos están compuestos por varios valores de entrada y el valor de salida esperado para dichas entradas, y son proporcionados por el maestro o supervisor. A partir de estos ejemplos, el algoritmo deberá aprender a predecir la salida para cualquier entrada, incluyendo aquellas que no haya visto nunca.
- **Aprendizaje no supervisado** o *unsupervised learning*: a diferencia del aprendizaje supervisado, en el aprendizaje no supervisado los ejemplos que conforman el conjunto de entrenamiento no contienen el valor de salida esperado, por lo que el algoritmo deberá aprender a realizar una clasificación de los datos proporcionados partiendo únicamente de los valores

de entrada.

- **Aprendizaje por refuerzo** o *reinforcement learning*: es una técnica basada en la psicología conductista, en la que el algoritmo aprende lo buena o mala que ha sido una acción tomada en función del resultado obtenido. Dicha acción no se valora comparándola con la salida esperada como ocurriría en el aprendizaje supervisado, sino que se le proporciona una recompensa que depende del estado al que conduzca esa acción. Si la acción le conduce a un estado objetivo la recompensa será positiva, mientras que si le conduce a un estado que debe evitar la recompensa será negativa. Por tanto, el algoritmo aprende a seguir una secuencia de acciones que le proporciona un valor máximo de recompensa. Este tipo de aprendizaje es especialmente útil cuando no se conoce cuál es la acción óptima que se debe tomar dado un estado concreto, pero sí se puede valorar si una acción, una vez tomada, ha sido positiva o negativa.

## 2.2. Reinforcement Learning. Q-Learning

Un niño trata de dar sus primeros pasos. Se pone de pie, avanza su pierna derecha y cae. La próxima vez, lo intenta con la izquierda, aguanta erguido un breve instante de tiempo y cae de nuevo al tratar de mover la derecha. El niño prueba distintas variantes, como dar pasos más cortos, flexionar más o menos las rodillas... y eventualmente, consigue dar varios pasos sin caerse y comienza a andar.

El niño no conoce las leyes de la mecánica de Newton ni el concepto de base de sustentación, y sin embargo es capaz de andar (como todos hicimos en algún momento de nuestras vidas). Esto nos lleva a pensar que, para aprender, tal vez no sea necesario un fundamento teórico sobre aquello que se quiere aprender, sino relacionar nuestras acciones con sus correspondientes consecuencias, adaptando nuestro comportamiento para llevar a cabo unas acciones y evitar otras.

Desde el punto de vista de la psicología (y en el contexto del aprendizaje), **reforzar** es el acto de fortalecer o incrementar la posibilidad de tomar una determinada acción en respuesta a una observación, porque se percibe un mayor beneficio al tomar esa acción que al tomar otras.

Al beneficio obtenido al tomar una acción se le denomina **recom-**

**pensa**, y es lo que guía el refuerzo de unas acciones frente a otras. Cuando a un perro se le da una galleta por completar una tarea, el perro asociará el hecho de completar la tarea con recibir la recompensa (en forma de galleta), lo que reforzará la decisión del perro de llevar a cabo las acciones que completan la tarea.

Esta forma de aprender a fortalecer la decisión de tomar una acción, motivada por la recompensa que se recibirá al tomar dicha acción, es a lo que llamamos **aprendizaje por refuerzo**, y es la forma en la que los niños aprenden a andar.

El niño aprende a andar basándose en las recompensas (mantenerse en equilibrio) y las penalizaciones (caerse) que obtiene a consecuencia de sus acciones, y no a través de haber establecido complejas relaciones entre cada acción y su resultado. El niño puede aprender a andar gracias a que obtiene *feedback* del mundo exterior, con el que es capaz de reforzar la realización de unas acciones frente a otras.

Y, ¿cómo se relaciona todo esto con la inteligencia artificial? Muchas de las técnicas de inteligencia artificial parten de áreas del conocimiento que estudian la inteligencia humana. En el caso del aprendizaje por refuerzo, está inspirado en la psicología conductista. Además, aplica conceptos propios de distintos campos de investigación, como la neurociencia, las matemáticas o la ya mencionada psicología, junto con, naturalmente, la informática (Figura 2.1)

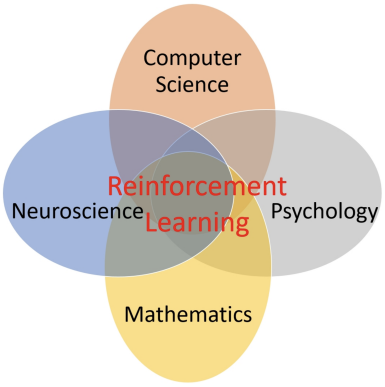


Figura 2.1: Campos de investigación relacionados con el aprendizaje por refuerzo. [38]

### 2.2.1. ¿Qué es el Reinforcement Learning?

Consideremos la tarea de enseñar a un perro a traer de vuelta una pelota cuando se le lanza lejos de su posición. Obviamente, no se le puede decir al perro qué tiene que hacer, es decir, no se le puede enseñar explícitamente a devolver la pelota (¡precisamente lo que queremos que aprenda!). En su lugar, simplemente lanzamos la pelota y, cada vez que la recoja y la traiga de nuevo a nuestra posición, le damos una galleta. El perro empezará a darse cuenta de qué acciones le llevan a recibir una galleta, y repetirá esas acciones.

A los sujetos del aprendizaje se les denomina agentes. Un *agente* es una entidad que actúa en base a las observaciones de su entorno. Cualquier ser vivo es un agente.

En software, un **agente** es un programa capaz de tomar decisiones y llevar a cabo acciones sobre un entorno basadas en las observaciones que obtiene del mismo.

El objetivo central del reinforcement learning es la creación de agentes inteligentes capaces de percibir su entorno y aprender a través de interactuar con el mismo. Para ello deben poder analizar su comportamiento y mejorarlo.

### 2.2.2. Elementos del Reinforcement Learning

En aprendizaje por refuerzo todo gira en torno a un *agente* tomando *decisiones* sobre un *entorno*. Tras estos conceptos abstractos, hay tres elementos básicos con los que se trabaja: *estados*, *acciones* y *recompensas*.

#### Estados

El *estado* es una descripción de la situación actual dentro de un problema. Si estuviésemos tratando de aprender a jugar al ajedrez, el estado sería la posición de las figuras en el tablero en un momento dado.

#### Acciones

Las *acciones* enumeran lo que el agente puede hacer en cada estado, por ejemplo, mover un alfil de `b2` a `d4` (pero no de `b2` a `c5`, pues ese movimiento no le está permitido a un alfil).

### Recompensas

La *función de recompensa* define como premia o penaliza el entorno las acciones que toma el agente. Generalmente, asocia pares estado-acción a valores numéricos que indican cómo de deseable para la resolución del problema es esa acción en ese estado. Un ejemplo de función de recompensa en el ajedrez sería otorgar 2 puntos por derrotar un peón rival, 5 por hacer lo propio con una torre, caballo o alfil, 10 por acabar con la reina, y 50 por ganar la partida (abatiendo al rey, por supuesto).

A pesar de utilizar el término recompensa, ésta no tiene por qué ser positiva. Cuando es positiva, se corresponde con la noción habitual que tenemos de recompensa, mientras que cuando es negativa se corresponde con una penalización (en ajedrez, perder una pieza podría tener una "recompensa" negativa).

Las recompensas pueden no ser inmediatas. Hay problemas que para alcanzar la solución exigen aparentemente *alejarse* (moverse a estados localmente peores). Ejemplo de este tipo de problemas sería el cubo de *Rubik*, donde si evaluamos la distancia a la solución como el número de cuadrados que no están bien posicionados, hay configuraciones del cubo que a pesar de "estar más lejos" según esta heurística, necesitan menos movimientos para llegar a la solución. Otro ejemplo sería el de escalar una montaña, donde puede ser necesario descender para alcanzar la cima.

En general, las *recompensas retardadas* se dan cuando no se puede seguir una estrategia voraz (tomar en cada momento la acción que más te acerca a la solución) para resolver el problema.

Por último, no todas las acciones han de tener una recompensa asociada (de hecho, no es una práctica habitual). Para la mayoría de problemas existirán acciones que no tienen una repercusión destacable en la resolución del mismo. Al fin y al cabo, el uso de recompensas sirve precisamente para destacar unas acciones sobre otras.

En reinforcement learning no se enseña al agente qué debe hacer o cómo debe hacerlo, sino que se le da una recompensa por cada acción que toma. Las acciones que conduzcan a la resolución del problema tendrán recompensas positivas, mientras que las que se alejen de la solución tendrán recompensas negativas. El agente debe descubrir qué acciones devuelven una mayor recompensa mediante prueba y error.

### 2.2.3. *Markov Decision Process*

La primera tarea al intentar resolver un problema (es decir, crear un agente que sea capaz de resolver el problema) mediante aprendizaje por refuerzo es definir los tres componentes básicos mencionados en la sección anterior, es decir: encontrar una buena representación para los estados, especificar cuáles son las acciones permitidas para cada estado y definir una función de recompensa.

Los conceptos anteriores se formalizan en lo que se denomina **Proceso de Decisión de Markov** (MDP, por sus siglas en inglés, *Markov Decision Process*), un *framework* matemático para modelar sistemas de toma de decisiones por parte de un actor/agente (o *decision maker* en la literatura original) en un entorno. Los elementos de un MDP (Figura 2.2)son:

- **S**: conjunto finito de estados en los que se puede encontrar el agente en un momento dado
- **A**: conjunto finito de acciones
- $\mathbf{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})$: función de transición de estados, que define la probabilidad de que el estado al que se llega si desde el estado $\boldsymbol{s}$ el agente selecciona (decide tomar) la acción $\boldsymbol{a}$ sea $\mathbf{s}'$
- $\mathbf{R}(\mathbf{r}|\mathbf{s},\mathbf{a})$: función de recompensa, que define qué recompensa $\boldsymbol{r}$ se obtiene si desde el estado $\boldsymbol{s}$ el agente selecciona la acción $\boldsymbol{a}$

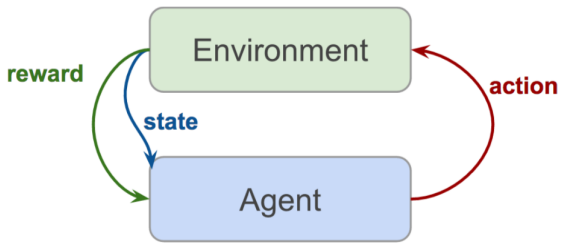


Figura 2.2: Visión general de un MDP. [12]

El agente interactúa con el entorno mediante acciones que le mueven de un estado a otro. Por cada acción que toma, recibe una recompensa. Una recompensa no es nada más que un valor numérico, por ejemplo, +1 por una buena acción y -1 por una mala acción. Si quisiéramos enseñar a nuestro agente a escapar de un laberinto, una "buena acción" podría ser dar un paso sin chocarse con los muros del laberinto y una "mala acción" sería chocarse contra ellos. Nótese que esta elección es totalmente arbitraria, pues si invertimos lo que entendemos por buena acción, podríamos obte-

ner un agente que esté continuamente chocandose contra el muro (algo bastante inútil).

Casi cualquier problema de aprendizaje por refuerzo puede ser modelado como un MDP, concretamente todo aquel que cumpla con la **asunción de Markov**: el siguiente estado *s'* depende únicamente del estado actual $s$ y de la acción $a$ que se tome en ese estado. Los MDPs se pueden utilizar tanto para modelar sistemas no deterministas, para los que la función de transición devuelve una distribución de probabilidad sobre los posibles estados a los que se puede llegar desde un estado dado, como deterministas, en donde el estado siguiente para un estado y acción dados es siempre el mismo. La función de transición sigue siendo una distribución de probabilidad, pero en este caso solo un estado tiene una probabilidad mayor que 0 (concretamente 1) de ser el estado siguiente.

Las recompensas obtenidas y los estados por los que pasa un agente a medida que toma acciones en un MDP forman un **episodio**. Un episodio es una serie de tuplas (estado, acción, recompensa). Cada una de estas tuplas diremos que es una **experiencia**. Los episodios finalizan al alcanzar un estado *terminal* (como podría ser el estado de *jaque mate* en el ajedrez).

Los MDP son útiles para modelar el proceso por el cual el agente aprende a realizar una determinada tarea. Por aprender entendemos obtener progresivamente una mayor recompensa, y para obtener una mayor recompensa el agente debe tomar mejores decisiones.

Las decisiones que toma un agente vienen determinadas por una función $\pi$ que dado un estado devuelve una acción

$$\pi(s) \rightarrow a$$

A esta función $\pi$ se le denomina **política**, porque determina cómo actúa un agente (que acción decide tomar) en base al estado en que se encuentra. Nuestro objetivo es entonces encontrar una política de actuación óptima para nuestro agente, que maximice la recompensa total obtenida en cada episodio.

Para ello, buscaremos aquella política que tome las decisiones que maximizan la *recompensa futura esperada* que, por no existir una traducción acordada en la literatura en castellano, llamaremos ***future return*** (el término utilizado en la literatura en inglés).

### Future return

Para decidir cuál es la mejor acción en un determinado estado hay que considerar no sólo la recompensa inmediata obtenida sino también las futuras recompensas que se pueden obtener desde el estado al que nos lleva esa acción. Es decir, hay que evaluar no sólo los efectos que provoca tomar esa acción sino sus consecuencias a largo plazo.

Por tanto, si queremos maximizar la recompensa total obtenida el agente debe seleccionar en cada paso la acción que, a la larga, le vaya a proporcionar mejores recompensas. Esta acción será:

$$max_a(r_a + R)$$

donde $r_a$ es la recompensa inmediata y $R$ la recompensa futura esperada. Hablamos de recompensa *esperada* porque, a priori, no sabemos cuál va a ser. A medida que el agente incorpore experiencias a su "memoria", podrá conocer de antemano la potencial recompensa que obtendrá al final del episodio si decide tomar esa acción en ese estado.

La definición del término $R$ es:

$$\sum_{k=t+1}^{T} r_k$$

es decir, si la acción $a$ se tomó en el instante $t$, $R$ es la suma de las recompensas obtenidas desde el instante $t+1$ hasta la terminación del episodio (instante $T$).

Si en cada paso nuestra política selecciona la acción que maximiza $r_a + R$, obtendríamos un agente sin ningún sentido de la urgencia por conseguir recompensas, lo que podría aplazar en exceso su obtención. Generalmente querremos forzar a nuestro agente a conseguir recompensas lo antes posible. Para ello, incluiremos un factor $\gamma$ en la expresión que controle la importancia que el agente da a las recompensas inmediatas frente a las futuras. Si desplegamos el término $R$, la expresión que nuestro agente debe maximizar para cada estado es:

$$return = r_a + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T-t} r_T$$

$\gamma$ toma valores entre 0 y 1, de modo que si $\gamma = 1$ se le da el mismo valor a las recompensas inmediatas que a todas las futuras

y si $\gamma = 0$ sólo se tienen en cuenta las recompensas inmediatas. Los valores típicos para $\gamma$ están en 0.90-0.99. La razón para elegir estos valores es que no todos los problemas se pueden dividir en episodios (sucesión finita de pasos que terminan en un estado final). Para un número infinito de pasos, con $\gamma = 1$ el valor para *return* diverge a infinito cualesquiera que sean las acciones elegidas. Con cualquier valor distinto de 1 para $\gamma$, las recompensas que se obtienen muy alejadas del estado final tienden a 0. Cuanto más se aleje $\gamma$ de 1, más pequeña es la "ventana de tiempo"que se le da al agente para obtener recompensas.

A $\gamma$ se le conoce comúnmente como ***factor de descuento***, porque descuenta el valor de la recompensa en función de cuando se obtenga (cuanto más tarde, más se descuenta).

### Value learning

La política de nuestro agente debe evaluar en cada estado las acciones que tiene disponibles y seleccionar de entre todas ellas la que considere mejor. Que una acción sea mejor que otra depende del criterio que se utilice para compararlas. En nuestro caso, evaluaremos el *future return* que ofrece cada acción y nos quedaremos con el máximo.

Esta técnica es conocida como **value learning** porque lo que se acaba obteniendo es un valor para cada par estado-acción que nos indica cómo de deseable es en un estado tomar una determinada acción. Con esta técnica, la tarea de aprender a montar en bicicleta sería la de asignar valores a cada ángulo de inclinación junto con las acciones que puedes tomar en esa posición.

Como se ha mencionado con anterioridad, el aprendizaje por refuerzo consiste esencialmente en aprender mediante prueba y error. Inicialmente, se desconoce cuál es el verdadero *future return* de cada acción en cada estado. A medida que el agente pruebe combinaciones de estados-acciones (i.e. experiencias) y vea qué recompensas arrojan, podrá ir modificando estos valores de manera que sean más verídicos.

Por tanto, para maximizar la recompensa total obtenida en cada episodio, los valores de *future return* deben corresponderse con lo que realmente se obtiene. Como serán estos valores los que determinen qué acción se toma en cada momento, el proceso de aprendizaje se puede resumir en aprender una asignación óptima de estos valores, de manera que en cada estado el agente sepa qué

acción es realmente la mejor.

### 2.2.4. Proceso de aprendizaje

Hemos visto que nuestro objetivo de conseguir que el agente maximice la recompensa obtenida pasa por asignar valores a cada par estado-acción que le informen cuál es la mejor acción en cada estado. Al proceso por el cual se pasa de tener unos valores aleatorios (o inicializados de cualquier otro modo que no aportan información realista) a una asignación que nos ayuda a tomar las decisiones que nos llevan a una mayor recompensa le conocemos como **proceso de aprendizaje**.

El proceso de aprendizaje consistirá en que el agente almacene experiencias sobre el dominio de aprendizaje (ajedrez, andar en bicicleta, salir de un laberinto) y utilice la información obtenida de las mismas para modificar sus valores.

Si desde un estado $s$ se toma una acción $a$ y $n$ pasos más adelante se obtiene una gran recompensa, habrá que modificar de alguna manera el valor del par $(s, a)$ para que, la próxima vez que el agente pase por ese mismo estado $s$ sepa que tomando la acción $a$ puede obtener buenas recompensas. Nótese que para obtener dicha recompensa habría que tomar nuevamente las $n$ acciones que distan desde que se toma la acción $a$ hasta que se obtiene la recompensa. Sin embargo, no queremos prefijar en nuestro agente una secuencia de acciones (carecería de interés todo este trabajo, pues eso ya sabemos hacerlo). Más bien queremos guiar al agente en sus decisiones para que sea capaz de anticipar su futuro. En este punto juega un papel fundamental el *factor de descuento*: cuanto más alto sea el valor de $\gamma$, mayor será el *campo de visión* hacia las futuras recompensas, pero menor será la precisión de las predicciones (el *future return* se puede entender como una "predicción" de cuánta recompensa nos espera).

El proceso de aprendizaje está repleto de variables conocidas como **hiperparámetros** que afectan notablemente al desarrollo de éste. Uno de ellos, el factor de descuento, ya lo hemos mencionado. Otro, muy decisivo también para el aprendizaje, es el *equilibrio exploración-explotación*.

### El dilema exploración-explotación

El agente puede *explorar* nuevas acciones que podrían llevarle a conseguir mejores recompensas, o puede *explotar* las acciones que ya conoce y que han resultado en una buena recompensa. Decimos que el agente **explora** cuando toma una decisión que nunca antes había tomado. Decimos que el agente **explota** (su conocimiento sobre el problema) cuando toma una acción para la cual ya conoce la recompensa que obtendrá porque ya la ha tomado antes.

Si el agente explora, es muy probable que las recompensas que obtenga no sean buenas, pues no se sustenta en ningún conocimiento para tomar sus decisiones, sino que las toma de manera aleatoria con la esperanza de eventualmente llegar a un estado favorable.

Si el agente sólo utiliza el conocimiento que ya tiene (explota), es probable que nunca descubra aquellas acciones que le conducirían a una mayor recompensa.

Encontrar un equilibrio entre cuanto explotar el conocimiento adquirido y cuanto explorar en busca de nuevo conocimiento es clave en el proceso de aprendizaje.

Para obtener la máxima recompensa posible, el agente preferirá tomar acciones que ya ha probado en el pasado y que han resultado ser efectivas. Sin embargo, para descubrir tales acciones tuvo que tomarlas en algún momento cuando aún eran desconocidas, ignorando su conocimiento sobre el problema y probando acciones que nunca antes había tomado.

Por tanto, el agente tiene que *explotar* las acciones que ya conoce para obtener una buena recompensa, pero tiene a su vez que *explorar* nuevas opciones para poder tomar mejores decisiones en el futuro. La complejidad en la decisión de cuánto explorar y cuánto explotar viene del hecho de que el aprendizaje se consigue mediante una combinación de ambas: el agente explora nuevas acciones y progresivamente favorece la explotación de aquellas que parecen mejores.

En la práctica, los niveles de exploración y explotación son dinámicos: al inicio del aprendizaje, cuando aún no se tiene ningún conocimiento sobre qué acciones son mejores, el nivel de exploración predomina sobre el de explotación. A medida que el agente descubre cuáles son esas acciones, la exploración empieza a disminuir progresivamente en favor de la explotación.

### 2.2.5. Q-Learning

El Q-Learning es una técnica de aprendizaje por refuerzo en la que se intenta que el agente aprenda una función: la función-Q, que para cada par estado-acción devuelve su calidad (***Q**uality*), esto es, como de deseable para la resolución del problema es tomar esa acción en ese estado. Teóricamente, con el uso de esta función, el agente podría conocer, para cada estado, cuál es la mejor acción que puede tomar (aquella para la cual la función devuelva un mayor valor) de entre todas las posibles para ese estado.

La función-Q se define como:

$$Q(s, a) \rightarrow q$$

El valor devuelto, $q$, representa la recompensa que esperamos alcanzar si, estando en el estado $s$ se toma la acción $a$ y, todas las sucesivas acciones tomadas desde $a$ hasta el estado terminal son las mejores acciones que se podrían tomar. Formalmente, si en el momento $t$ el agente se encuentra en el estado $s$, tomando la acción $a$, la recompensa máxima esperada (o *future return*) es:

$$Q^*(s, a) = max(\sum_{i=t}^{T} \gamma_i r_i)$$

donde $\gamma_i$, $r_i$ son respectivamente el *factor de descuento* y la recompensa obtenida de la $i$-ésima acción, y $T$ es el número total de estados por los que pasa el agente.

Diferenciamos $Q$ de $Q^*$: $Q$ es una función que para cada par estado-acción devuelve un valor que indica la calidad del mismo, y $Q^*$ es la función óptima de entre todas las posibles asignaciones de valores a pares estado-acción (i.e. funciones-Q). Recordemos que el objetivo del Q-Learning es aprender esta función óptima.

¿Cómo podemos entonces aprender esta función? Por conveniencia, podemos ver la función como una tabla, con una entrada por cada combinación estado-acción. El objetivo es entonces rellenar esta tabla, que llamaremos tabla-Q, con valores apropiados (informativos). Necesitamos una regla para ir actualizando los valores de la tabla a partir de las experiencias del agente.

### La ecuación de Bellman

Para que el agente tenga en cuenta las recompensas a largo plazo definiremos los valores-Q en función de los valores-Q futuros. La **ecuación de Bellman** establece que la máxima recompensa esperada por tomar la acción $a$ en el estado actual $s_t$ es la recompensa inmediata $r$ más la máxima recompensa esperada por tomar la siguiente acción $a'$ en el siguiente estado $s_{t+1}$

$$Q^*(s,a) = r_t + \gamma max_{a'} Q^*(s_{t+1}, a')$$

A partir de esta expresión recursiva, podemos definir una regla para actualizar los valores-Q pasados a partir de los futuros. La dinámica es la siguiente: tras alcanzar un estado terminal, digamos $S_n$, tendremos un valor-Q correcto, el de la acción tomada en el estado inmediatamente anterior a $S_n$, que será igual a la recompensa obtenida $r_n$ pues no hay estado siguiente (y por tanto tampoco recompensas futuras). Ahora, podemos propagar este conocimiento que sabemos que es correcto hacia atrás de la siguiente manera

$$Q_t \leftarrow Q_{t+1} \leftarrow \ldots \leftarrow Q_n$$

En cada paso, sobrescribiremos el valor-Q de cada par estado-acción por los que pasa el agente según la expresión

$$Q(s_i, a_i) = (1-\alpha) * Q(s_i, a_i) + \alpha * (r_i + \gamma * max_a Q(s_{i+1}, a))$$

donde introducimos un nuevo hiperparámetro, $\alpha$, conocido como ***learning rate*** (tasa de aprendizaje), que al igual que el factor de descuento toma valores entre 0 y 1. En cada paso, el nuevo valor-Q será el valor que ya se tenía más la información que propagan las experiencias futuras. El *learning rate* controla en cada actualización cuanta información se toma de lo que ya se conoce y cuanta información nueva se incorpora. Para $\alpha = 0$ no hay aprendizaje, pues en cada paso $Q$ tomará el valor que ya tenía. Para $\alpha = 1$, en cada actualización se desecha el conocimiento previo y se reemplaza por el de las nuevas experiencias.

El principal problema del Q-Learning es que sólo es viable para problemas con un espacio de estados-acciones reducido. Como hemos visto, los valores de la función-Q los almacenamos en una tabla. Para los problemas más interesantes, esta tabla puede llegar

a ser extremadamente grande, hasta el punto en que no pueda siquiera almacenarse en memoria. Como veremos más adelante, una de las soluciones a esta problemática es aproximar la función-Q con una red neuronal, que será la que abordaremos en este proyecto.

## 2.3. Deep Learning. Redes Neuronales

El **Deep Learning** es un subcampo del Machine Learning que se sirve de varias capas de Redes Neuronales para encontrar soluciones al problema que se le plantea. Aunque se considera que el Deep Learning surgió en 2006, hasta 2012 fue un campo de la IA ignorado por gran parte de los desarrolladores. Tiene su resurgimiento cuando se comienza a usar el Deep Learning para reconocimiento de voz o speech recognition y alcanza gran fama en 2016, momento en el que el programa AlphaGo de Google DeepMind derrota a Lee Sedol, jugador profesional coreano de Go.

Gran parte de los conceptos básicos del Deep Learning surgieron en los años 80 y 90, pero la década de 2010 resulta propicia para su desarrollo debido a diferentes factores. Los dos más determinantes son la aparición de grandes conjuntos de datos etiquetados de gran calidad y la capacidad de ejecución paralela de las GPU.

El Deep Learning resulta sumamente útil a la hora de reconocer patrones en una gran cantidad de datos. Una vez extraídos estos patrones, el sistema será capaz de utilizarlos para etiquetar nuevos datos de entrada. Por ejemplo, si el programa recibe fotografías e indicamos que en ellas aparece el primer ministro de Francia, el programa acabará aprendiendo un patrón para reconocer su cara. Al final, si recibe una foto en la que aparece el primer ministro francés, pero no se lo indicamos, el programa será el que etiquete dicha foto como una foto en la que aparece el primer ministro.

Para realizar este proceso, el modelo recibirá un conjunto de datos de ejemplo. Cuanto mayor sea este conjunto, mayores serán tanto la cantidad de patrones que detecte como su precisión.

Las **Redes Neuronales** son un modelo computacional basado en el funcionamiento del cerebro y diseñado para el reconocimiento de patrones. Estos patrones son patrones numéricos que estarán contenidos en vectores, y todo dato recibido como entrada, ya sean imágenes, sonido, texto, etc., será transformado en dichos vectores. Los datos, una vez traducidos, serán interpretados por la red neuronal y se etiquetarán o agruparán.

Tradicionalmente las redes neuronales sólo estaban compuestas por una capa de entrada (*input*), una capa oculta (*hidden layer*) y una capa de salida (*output*). Esto suponía que una gran parte del trabajo de los desarrolladores consistiera en asegurarse de que la capa de entrada recibiera los datos en un formato adecuado, destacando las características que pudieran ser relevantes para el aprendizaje de la red neuronal, permitiéndole extraer patrones. Este proceso se llama *feature engineering*.

Para agilizar el proceso de desarrollo se añadieron una o más capas ocultas que consiguen realizar parte o todo el *feature engineering* por sí mismas. Esto proporcionaba a la red más profundidad (*deep*). Se considera Deep Learning a cualquier red neuronal con mínimo cuatro capas, una de entrada, otra de salida y, entre medias, dos o más capas ocultas (*hidden layers*).

Para que la red neuronal sea capaz de predecir cómo etiquetar o clasificar datos de forma precisa necesita ser entrenada con datos ya etiquetados. Por ello en el Deep Learning se necesita una gran cantidad de datos de ejemplo.

### 2.3.1. Estructura de las redes neuronales

Las redes neuronales están compuestas por capas y, a su vez, dichas capas están compuestas por nodos, también llamados neuronas.

Los nodos tienen la siguiente estructura:

- Una o varias entradas que pueden recibir los datos originales que se quieren procesar o la salida de otro nodo.
- Los pesos dados a cada entrada del nodo. Los pesos amplifican o reducen la entrada según lo que el modelo quiera aprender.
- La función de suma que se encarga de sumar todas las combinaciones de peso-entrada de la función.
- La función de activación que determina si el nodo estará activo o no según el valor obtenido en la función suma. Esto significa que determina lo que el nodo devuelve según el valor calculado y el tipo de función de activación. Ejemplos de función de activación son los siguientes:
  - Función escalón: devuelve 0 si el valor es negativo y 1 si es positivo.
  - Función lineal: devuelve directamente el valor calculado.
  - Función lineal a tramos: devuelve el valor calculado si está dentro de ciertos límites, por ejemplo, la función lineal

ReLU devuelve el valor calculado si es positivo y 0 si es negativo

La estructura de un nodo o neurona sería como se ve en la figura 2.3.

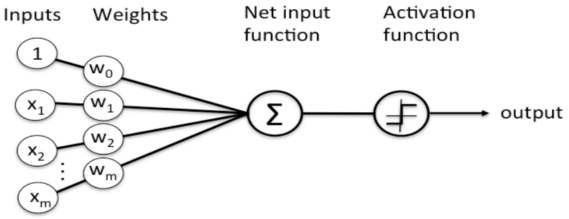


Figura 2.3: Nodo/Neurona. [11]

Como se indica anteriormente, la red está organizada en capas, filas de nodos. Cada capa recibe datos de entrada de la capa anterior y su salida se transmite a la entrada de la siguiente, como se ve en la figura 2.4.

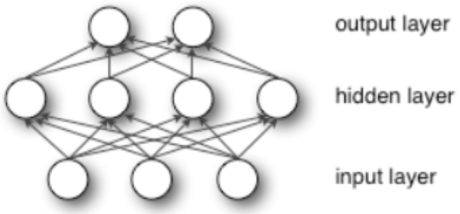


Figura 2.4: Capas de una red neuronal. [11]

Cada capa se encarga de procesar distintos atributos de los datos de entrada de la red neuronal o atributos de los datos procesados por la capa anterior. A su vez, según se profundiza en las capas de la red más aumenta la complejidad y la abstracción de los atributos que se procesa. Se ve en el ejemplo de la figura 2.5.

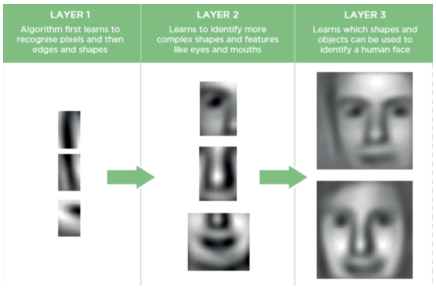


Figura 2.5: Ejemplo de uso de las capas para la partición del problema. [4]

### 2.3.2. Definición del modelo

Entendiendo como modelo la red neuronal en su completitud, se procederá a describir los aspectos que lo conforman para tener una idea general de su composición y funcionamiento.

#### Parámetros

Como primer paso para crear nuestro sistema de Machine Learning, definiremos las etiquetas (*labels*), los atributos (*features*) y los ejemplos:

- **Etiquetas/Labels**: es el valor que estamos prediciendo. En el caso del problema del ministro francés serían las coordenadas del rectángulo que encuadra su cara en la foto.
- **Atributos/Features**: es una variable de entrada. El número de variables de entrada puede variar dependiendo del problema que se está tratando

  $$X = x_1, x_2, \ldots, x_N$$

  En el caso del problema ministerial podría ser el color de todos los píxeles de la foto.
- **Ejemplos**: son una instancia particular de datos **X**. Los ejemplos pueden ser de dos tipos:
  - Etiquetados (*labeled*): estos ejemplos son los utilizados para entrenar al modelo. Son la combinación de los atributos y su etiqueta. Recibimos la foto del ministro con las coordenadas de su cara.
  - Sin etiquetas (*unlabeled*): contiene los atributos, pero no la etiqueta. Son los datos que recibe el modelo para hacer predicciones sobre ellos, devolviendo la etiqueta que deberían tener. Recibimos la foto, pero esta vez no tenemos las coordenadas de la cara.

El modelo define la relación entre atributos y etiquetas.

Un modelo tiene dos fases:

- **Entrenamiento**: creación o proceso de aprendizaje del modelo. Mediante ejemplos etiquetados enseñamos al modelo la relación entre atributos y etiquetas.
- **Inferencia**:el modelo entrenado hace predicciones sobre ejemplos sin etiquetas. Recibe los atributos del ejemplo y él infiere una etiqueta ($y'$).

Existen dos tipos principales de modelos:

- El modelo de **regresión**, que predice valores continuos y pueden ser de regresión lineal y no lineal.
- El modelo de **clasificación**, que predice valores discretos.

Una neurona computa una combinación lineal de sus entradas y luego aplica una función de activación no lineal para construir un modelo no lineal. Por ello procederemos a explicar el modelo de regresión lineal.

### Regresión lineal

La ecuación que sigue un modelo en el que sus atributos y sus etiquetas tienen relación lineal es la siguiente:

$$y' = b + w_1x_1 + \ldots + w_nx_n$$

donde

$y'$ es la etiqueta predicha
$b$ a intersección con el eje $y$ (la ordenada al origen). También llamado *bias* o $w_0$
$w_i$ el peso del atributo $i$. Es la pendiente de una ecuación lineal.
$x_i$ es el atributo i.

Un modelo que sólo tiene un atributo sería de la forma: $y' = b+w_1x_1$ con lo cual sería igual que una ecuación lineal $y = mx+b$, donde:

$y$ es el resultado
$x$ es el parámetro de entrada
$m$ es la pendiente.
$b$ es la intersección con el eje y

El ejemplo propuesto es el siguiente:

Existe una relación entre la frecuencia con la que los grillos cantan y la temperatura de su ambiente y queremos que nuestro modelo, al recibir esta frecuencia, prediga la temperatura. Usaremos la frecuencia de canto de los grillos como atributo y será nuestra coordenada $x$. La etiqueta correspondiente será la temperatura en Celsius y será nuestra coordenada $y$. Tras recolectar datos obtenemos el resultado mostrado en la figura 2.6.

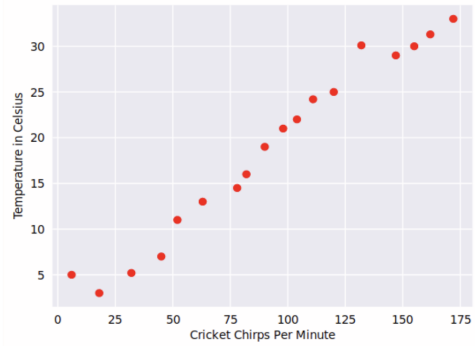


Figura 2.6: Chirridos por minuto vs. Temperatura en grados Celsius. [18]

Nuestro modelo generaría una ecuación lineal que se aproximara a los resultados de la figura 2.7.

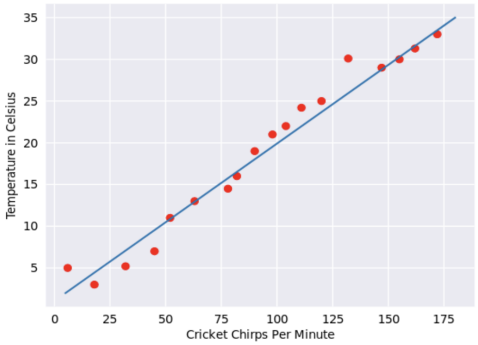


Figura 2.7: Relación lineal. [18]

### Entrenamiento y reducción de la pérdida de una neurona

Cuando se entrena un modelo con ejemplos etiquetados lo que ocurre internamente es una modificación de los pesos (incluido el bias) para minimizar una *función de loss o pérdida*. La función recibe las predicciones del modelo ($y'$) para determinados atributos ($x$) y las etiquetas correctas ($y$) correspondientes a esos atributos. La pérdida indica lo incorrecta que fue una predicción (pérdida $= |y - y'|$).

Para conocer la pérdida general del modelo se pueden utilizar varias funciones de pérdida, una de las más comunes es la MSE (*Mean Square Error*) que se corresponde con la media de las pérdidas elevada al cuadrado.

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediccion(x))^2$$

Los pesos están inicializados con un valor cualquiera y van modificándose hasta que el modelo converge. Un modelo converge cuando la pérdida general deja de cambiar o cambia muy lentamente.

Ejemplo en la figura 2.8 de pérdida en dos modelos diferentes, siendo las flechas rojas la pérdida del modelo.

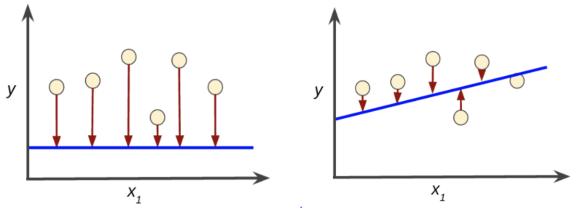


Figura 2.8: Pérdida alta en el primer modelo; Pérdida baja en el segundo modelo. [18]

Se puede apreciar que en el segundo modelo la pérdida es menor.

Para ajustar los pesos se utiliza un **algoritmo de optimización**, algoritmos que ayudan a reducir la función de pérdida. Existen varios tipos de algoritmos de optimización, siendo las dos categorías principales las siguientes:

- Algoritmos de optimización de primer orden. Utilizan el **gradiente** (vector que indica la dirección de máximo crecimiento de la función en un punto) sobre la función de pérdida calculado a partir de derivadas parciales. El más representativo es el **descenso de gradiente**.
- Algoritmos de optimización de segundo orden. Al usar derivadas parciales de segundo orden (cuyo cálculo es bastante costoso) para saber cómo minimizar la pérdida, son menos utilizadas.

Para poder hacerse una idea de cómo se reduce la pérdida se explicará cómo realiza el proceso de optimización el descenso de gradiente.

Se sigue el siguiente proceso:

1. En la figura 2.9 se elige un valor inicial para el peso, en este caso $w_1$. La elección del valor inicial no es muy importante, por lo que se eligen valores al azar o se inicializa directamente a 0.

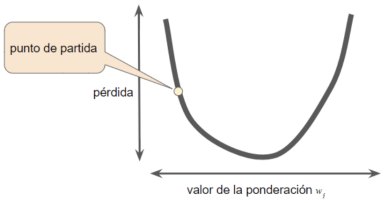


Figura 2.9: Elección de un punto de partida. [18]

2. En la figura 2.10 el algoritmo calcula hacia qué dirección de valores de $w_1$ se debe dirigir para acercarse al mínimo de pérdida. El tamaño del paso es determinado por la tasa de aprendizaje.

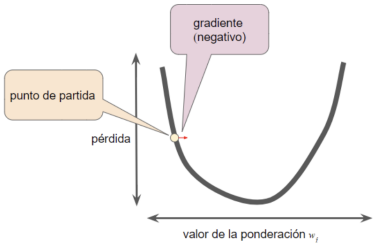


Figura 2.10: Descenso de gradiente. [18]

3. En la figura 2.11 se determina el siguiente valor de $w_1$.

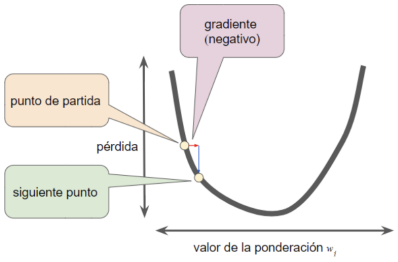


Figura 2.11: Siguiente valor de $w_i$. [18]

4. Se repite el proceso desde el paso 2 acercándose cada vez más al mínimo.

De esta forma se reduce cada vez más la pérdida hasta que converge y no puede encontrar un valor más pequeño o se alcanza el valor deseado.

Este proceso sería igual con varios pesos, cambiando las dimensiones de los vectores. Por ejemplo, para una neurona con dos pesos la pérdida tendría una representación tridimensional como en la

figura 2.12.

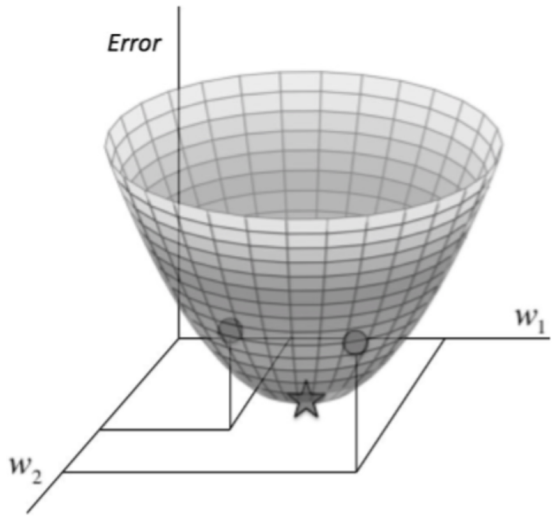


Figura 2.12: Superficie para un error cuadrático. [12]

#### 2.3.2.1. Backpropagation

Para entrenar las neuronas de todas las capas de una red neuronal el algoritmo más utilizado es *backpropagation*.

No se puede saber lo que deberían estar haciendo las neuronas de las capas ocultas, pero sí se puede calcular cómo de rápido cambia la función de pérdida si se cambia la actividad de una neurona de la capa oculta, si se modifica alguno de los pesos de alguna de sus conexiones. Se trata de encontrar el camino que más rápido haga decrecer la función de pérdida. En eso consiste *backpropagation*.

Cuando se le suministra un ejemplo de entrenamiento a una red con más de una capa de neuronas (esto es, con al menos una red oculta), la salida de cada capa está conectada con la entrada de la capa siguiente. De esta manera, los datos fluyen por la red hasta alcanzar la última capa (o capa de salida). La salida generada por la red se compara entonces con la salida esperada y se calcula la pérdida. La pérdida se *propaga hacia atrás*, desde la capa de salida hacia todas las neuronas de las capas anteriores. Las neuronas de las capas más alejadas de la capa de salida recibirán una fracción del la pérdida, correspondiente a la contribución relativa que cada neurona aporta a la pérdida.

Como la variación de los pesos influye en la función de pérdida,

se expresa como la derivada parcial de cada peso con respecto a la pérdida.

$$\frac{\partial E}{\partial w_k}$$

### 2.3.3. Grandes proyectos basados en Deep Learning

Hoy en día el Deep Learning forma parte de infinidad de proyectos, pero donde se está centrando gran parte de la atención es en el campo del reconocimiento facial y la extracción de datos de imágenes, denominado *computer vision*.

Este campo, *computer vision*, está siendo muy utilizado en la creación de jugadores automáticos de videojuegos para la extracción del estado del juego a partir del último o los últimos *frames* capturados.

Dos ejemplos de grandes proyectos que utilicen el *computer vision* son Google Vision [35] y Amazon Rekognition [9], descritos a continuación.

#### Google Vision

Google proporciona una API para desarrolladores capaz de extraer información que considera relevante de imágenes.

No sólo usa reconocimiento facial, también es capaz de reconocer el estado de ánimo de las personas que detecta y su detección de objetos es muy avanzada. Tiene un conjunto de objetos reconocibles inmenso y puede reconocer elementos turísticos, como puede ser la Torre Eiffel. Además, aprende de los nuevos datos que se le aporten, por lo que el conjunto de objetos reconocibles no deja de crecer y la precisión es cada vez mayor.

Es capaz de extraer el texto que se encuentre en la imagen y funciona junto a la identificación automática de idiomas de Google.

Todo lo detectado puede ser filtrado automáticamente como contenido inapropiado.

Es una herramienta muy versátil, de gran fiabilidad y que se puede explotar para ser utilizada en infinidad de aplicaciones.

**Amazon Rekognition**

Al igual que Google Vision, se trata de una API con reconocimiento facial, de objetos, texto y contenido inapropiado, pero, además de en imágenes, también funciona con vídeos en tiempo real.

Además, reconoce escenas y la actividad que se está desarrollando en ellas.

Amazon pone como ejemplos de uso de la aplicación la verificación de usuarios, la contabilización de personas y su aplicación en la seguridad pública.

## 2.4. Deep Reinforcement Learning

Como hemos visto en secciones anteriores, el Q-Learning utiliza la tabla-Q como método de representación para la función-Q. Este método permite representar, de una manera muy precisa, el valor de las recompensas a largo plazo que se pueden obtener desde cada estado para cada una de las acciones disponibles.

No obstante, esta representación también implica un problema muy importante: aumentar la complejidad del estado también aumenta el tamaño de la tabla-Q, pero de un modo exponencial.

Esto puede verse fácilmente mediante un ejemplo. Imaginemos un juego muy simple en el que el jugador maneja a un aventurero que debe recoger varios tesoros para completar su misión. Ahora, queremos añadir un nuevo tesoro opcional que, de haberlo recogido al terminar la partida, nos hará ganar mucha más puntuación. Para ello, añadiremos un nuevo campo binario al estado, que nos indicará si ese tesoro ha sido o no recogido.

Como la tabla-Q debe tener una entrada para cada par estado-acción, ahora el tamaño de la tabla será el doble que el que tenía antes de la modificación, ya que todos los estados que ya existían anteriormente ahora tienen dos versiones; una en la que el nuevo tesoro aún no ha sido recogido y otra en la que sí lo ha sido.

Si añadimos nuevas modificaciones como la descrita anteriormente el tamaño de la tabla volvería a doblarse, dando lugar, como hemos dicho antes, a un crecimiento exponencial.

Por lo tanto, si nuestro objetivo es crear una inteligencia artificial capaz de resolver un problema complejo, podríamos encontrarnos con el impedimento de que el tamaño de la tabla-Q sea tan grande

que no sea posible almacenarla en memoria.

Pero ¿y si en vez de utilizar una representación de la función-Q tan explícita como la tabla-Q utilizásemos una aproximación de dicha función mediante el uso de una red neuronal? Precisamente esa idea tuvo el equipo de DeepMind en 2013 para llevar a cabo su proyecto *Playing Atari with Deep Reinforcement Learning* [31].

### 2.4.1. Deep Q-Networks

En este proyecto desarrollaron un agente de inteligencia artificial capaz de aprender a jugar a siete juegos de la consola Atari utilizando como estado los píxeles de la pantalla, tras un pequeño preprocesamiento en el que se convertían a una escala de grises y se reescalaba el tamaño a 110x84 píxeles, en lugar de los 210x160 originales (Figura 2.13). Los resultados obtenidos fueron impresionantes, siendo capaz de superar a un jugador humano experto en tres de estos juegos. Ellos lo llamaron *Deep Q-Network* (o DQN), y fue la primera implementación de un agente de Deep Reinforcement Learning.

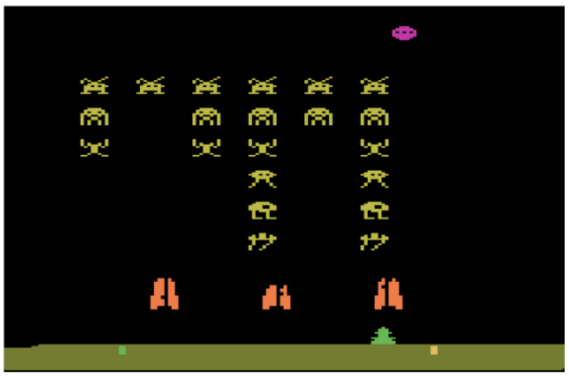

Figura 2.13: Captura del juego Space Invaders de la consola Atari. [31]

Como ya hemos dicho, su principal diferencia con respecto a un agente de Q-Learning habitual era la sustitución de la tabla-Q por una red neuronal que aproximara los valores de la función-Q para cualquier par estado-acción.

Esta red debía ser entrenada para que dicha aproximación fuese lo más precisa posible, por lo que el objetivo era reducir al mínimo la diferencia entre el valor aproximado por la red para un par estado-acción y su recompensa futura esperada:

$$min([Q(S_i, a_i)] - [r_i + \gamma * maxQ(S_{i+1}, a)])$$

Para agilizar este proceso añadieron otro pequeño cambio. A diferencia de la tabla-Q, que devuelve el valor de la función-Q para un único par estado-acción, a la hora de consultar a la red neuronal utilizaron como única entrada el estado, de modo que la red devolvía el valor de la función para todas las acciones posibles. De esa manera, para obtener el valor máximo de la función para todos los pares estado-acción en los que el estado se correspondía con el estado siguiente, era necesaria una única consulta a la red neuronal, lo cual era muy conveniente ya que el número de acciones posibles podía ascender hasta 18.

Por último, ya que las Deep Q-Networks no dejan de ser un MDP, las acciones tomadas dependen únicamente del estado actual. Esto suponía algún problema en juegos como *Breakout* (Figura 2.14), en los que algo tan importante como la velocidad de la pelota era imposible de apreciar teniendo en cuenta únicamente el último frame. Por tanto, decidieron que el estado estuviera formado por los cuatro últimos frames observados, de modo que se pudieran tener en cuenta los cambios que se producían en el entorno a lo largo del tiempo.

Figura 2.14: Captura del juego *Breakout* de la consola Atari. [31]

### 2.4.2. Estabilidad del aprendizaje. Problemas y soluciones

Un problema que se pudo observar era que la utilización de la red para predecir tanto el valor de la función-Q para el estado actual como para el estado siguiente hacía que el proceso de aprendizaje fuera muy inestable, ya que se dependía en gran medida de los valores aprendidos por la red y éstos cambiaban de manera constante con cada una de las actualizaciones.

Por otro lado, aprender de muestras de datos consecutivas es bastante ineficiente, ya que suelen tener un alto grado de correlación entre sí. Esto supone un gran inconveniente ya que nuestro objetivo es que los conjuntos de entrenamiento sean lo más representativos posible de todo el problema.

Para solucionar estos problemas, se llevaron a cabo las soluciones descritas a continuación.

### Target Q-Network

Para reducir el nivel de dependencia con los valores actuales de la red neuronal se introdujo una segunda red llamada "red de objetivos" o "*target network*", la cual era una copia de la red original. La primera red se utilizaba para predecir el valor de la función-Q para el estado actual, mientras que la segunda se utilizaba para predecir el valor del estado siguiente. A diferencia de la red original, esta "red de objetivos" no se actualizaba de manera constante, sino que igualaba el valor de sus parámetros a los de la red original después de que ésta se hubiera entrenado con varios conjuntos de entrenamiento.

La presencia de esta red proporcionaba estabilidad a los valores predichos para los estados siguientes, lo que a su vez proporcionaba también estabilidad para el aprendizaje.

### Experience Replay

Para que el entrenamiento no se realizara con muestras de datos consecutivas utilizaron un método llamado *experience replay*, consiguiendo reducir la correlación entre los datos y minimizar la posibilidad de que el aprendizaje se quedase atascado en un mínimo local. Consistía en almacenar en cada paso tuplas de la forma $(s, a, r, s')$ donde $s$ es el estado actual, $a$ la acción tomada, $r$ la recompensa obtenida y $s'$ el siguiente estado; y a la hora de entrenar la red se utilizaba un conjunto de entrenamiento formado por varias de estas tuplas seleccionadas de manera aleatoria de entre todas las almacenadas. De esta manera, la red podía entrenar utilizando muestras en un orden distinto en el que ocurrieron.

El número de muestras almacenadas era limitado, de modo que cuando se alcanzaba el tamaño máximo las más antiguas eran sustituidas por las nuevas, de manera que el conjunto estaba siempre actualizado.

Además, la posibilidad de utilizar una misma muestra en varios conjuntos de entrenamiento aumentaba en gran medida la eficiencia de los datos obtenidos.

### 2.4.3. Mejoras de las Deep Q-Networks

Pese a que los resultados de las DQN fueron realmente positivos, el equipo de DeepMind ha seguido trabajando desde 2013 con el fin de mejorar los puntos débiles que éstas presentaban, dando lugar a nuevas tecnologías.

#### Deep Recurrent Q-Networks (DRQN)

Recordemos que el equipo de DeepMind utilizó los cuatro últimos *frames* del juego como estado con el fin de tener en cuenta los cambios que se producían en el entorno a lo largo del tiempo. Pese a obtener buenos resultados, este número podría no ser el óptimo para todos los problemas.

Como solución decidieron utilizar una red neuronal recurrente, las cuales permiten aprender, a partir de su propio proceso de aprendizaje, comportamientos que evolucionan a lo largo del tiempo de una manera dinámica.

De este modo, la red aprendía durante el proceso de entrenamiento cuál era el número óptimo de frames a utilizar, a descartar aquellos que no contenían información relevante e incluso a tener en cuenta información bastante antigua.

Una evolución posterior de las DRQN permitió que aprendieran incluso a focalizar su atención en ciertas partes de la imagen, lo que supuso un importante aumento del rendimiento.

En comparación con las DQN, las DRQN demostraron obtener mejores resultados en juegos de disparos en primera persona y en algunos juegos en los que los hechos que se daban lugar en la partida tenían influencia a largo plazo.

#### Asynchronous Advantage Actor-Critic Agent (A3C)

Como el propio nombre indica, el A3C es un agente asíncrono, lo cual le permite ser lanzado en varios hilos de manera simultánea. Esta característica tiene como principal consecuencia un aumento muy destacable en la velocidad de entrenamiento.

No obstante, hay otra importante ventaja en que este agente se ejecute en varios hilos de manera simultánea: las muestras de datos obtenidas provienen de distintos escenarios, por lo que la correlación entre ellas disminuye en gran medida.

Una muestra de la clara mejora de rendimiento de esta tecnología es el tiempo que un agente A3C tarda en aprender a jugar al juego *Breakout* de la consola Atari (aproximadamente unas doce horas) en comparación con un agente DQN (entre tres y cuatro días).

#### UNsupervised REinforcement and Auxiliary Learning (UNREAL)

UNREAL es una evolución del A3C que tiene como finalidad obtener una buena representación del entorno en el que el agente será entrenado.

En el aprendizaje por refuerzo, todo el conocimiento adquirido se obtiene a través de las recompensas; pero es complicado predecir cuándo se obtendrán y cuál será su valor, lo cual convierte el aprendizaje en una tarea difícil.

Cuanto mayor sea el conocimiento del entorno más fácil será realizar estas predicciones, y por tanto más sencillo será realizar el proceso de aprendizaje.

Para ello, UNREAL añade una nueva tarea que consiste en aprender cómo las acciones tomadas por el agente afectan al entorno, independientemente de si se obtienen recompensas o no.

Para comprobar la calidad de la representación, UNREAL trata de predecir cuál será el valor de la próxima recompensa obtenida, de modo que si esta predicción es correcta es probable que dicha representación sea bastante acertada.

Estas mejoras han permitido que un agente UNREAL aprenda unas diez veces más deprisa que un agente A3C, lo que demuestra la importancia de tener una buena representación del entorno.

### 2.4.4. Costes. Tiempo y recursos hardware

Como hemos mostrado, el Deep Reinforcement Learning proporciona múltiples ventajas, pero también tiene una parte negativa centrada principalmente en los costes temporales del entrenamiento y en el coste de los recursos hardware necesarios para llevarlo a cabo.

El tiempo necesario para que nuestro agente aprenda a resolver un problema es directamente proporcional a la complejidad del mismo, siendo necesario en algunos casos el uso de recursos de cómputo muy potentes. DeepMind [21] requirió 64 CPUs durante más de 100 horas para que un agente aprendiera a controlar los movimientos de un cuerpo articulado como el que se muestra en la figura 2.15 de manera que fuera capaz de avanzar sobre un terreno con obstáculos sin caerse.

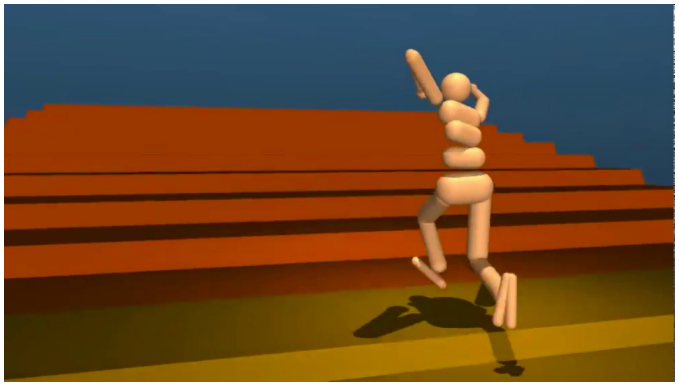

Figura 2.15: El agente trata de aprender a subir unas escaleras sin caerse. [21]

Además, para un aprendizaje satisfactorio, es necesario contar con una gran cantidad de ejemplos de entrenamiento con los que el agente pueda aprender. Para algunos problemas (los más interesantes), el espacio de posibles experiencias (en un estado tomar una acción y ver qué recompensa genera) puede ser inmenso.

En su primera versión, la red neuronal que utilizó DeepMind para aprender a jugar a *Breakout* utilizaba como entrada 9240 píxeles (correspondientes a la resolución adaptada de la pantalla, 110x84) en una escala de grises. Para una gama de grises discreta con k tonalidades, el número de estados sería de $k^{9240}$. Como para cada estado hay tres posibles acciones, y suponiendo que para un par estado-acción siempre se obtiene la misma recompensa, el espacio de posibles experiencias sería de $3k^{9240}$. A DeepMind le llevó 83 horas de juego obtener un conjunto de experiencias representativo que poder suministrar a la red neuronal para su aprendizaje.

No obstante, el desarrollo de mejoras como las mencionadas DRQN, A3C y UNREAL han permitido reducir tanto el tiempo de aprendizaje como los recursos hardware necesarios, aunque éstos siguen siendo elevados y no están al alcance de todo el mundo.

### 2.4.5. Otros proyectos destacados

A pesar de tratarse de una tecnología muy reciente, el Deep Reinforcement Learning ya cuenta con varios proyectos que han proporcionado resultados extremadamente positivos, siendo los más destacados AlphaGo [5] y OpenAI Dota 2 [24].

#### AlphaGo

Tras la adquisición por parte de Google de la compañía DeepMind en el año 2014, ésta volvió a sorprender al mundo en 2016 con su proyecto AlphaGo, el primer agente de inteligencia artificial capaz de derrotar a un jugador profesional del juego de tablero Go.

#### El juego del Go

El Go es un juego de estrategia proveniente de China en el que dos jugadores deben colocar piedras blancas o negras en un tablero con el fin de controlar más territorio que el rival. Una vez una piedra ha sido colocada no puede volver a moverse. Además, si un conjunto de piedras del mismo color es rodeado completamente por piedras del adversario, éstas son capturadas y eliminadas.

Las dimensiones del tablero son por lo general de 19x19 (Figura 2.16), dando lugar a más de $10^{170}$ configuraciones posibles para la partida y convirtiendo al Go, a pesar de la simplicidad de sus reglas, en un juego extremadamente más complejo que otros juegos similares, como por ejemplo el ajedrez.

Figura 2.16: Fotografía de una partida de go. [15]

### Tecnología utilizada por AlphaGo

La técnica de aprendizaje utilizada por AlphaGo combina la búsqueda avanzada en árboles mediante el método Monte Carlo con el uso de dos redes neuronales, las cuales utilizan una descripción del estado del tablero como entrada. La primera red, llamada "red de política" o "policy network", es la encargada de determinar cuál es el mejor movimiento a realizar en cada momento de la partida; mientras que la segunda, llamada "red de valores" o "value network" se encarga de predecir el ganador de la misma.

Por su parte, el método Monte Carlo utiliza un gran número de simulaciones de la partida para determinar cuál es la acción más beneficiosa en cada momento, basándose en el número de simulaciones terminadas en victoria a las que conduce cada una de ellas.

El entrenamiento de AlphaGo estuvo dividido en dos fases. En un primer momento fue entrenado mediante el uso de partidas reales en las que se enfrentaban jugadores humanos entre sí, con el fin de adquirir un estilo de juego similar al de éstos. Una vez completada esta fase pasó a entrenarse enfrentándose a sí mismo durante miles de partidas, utilizando el Deep Reinforcement Learning como método para aprender a partir de dicha experiencia.

### Resultados de AlphaGo

En su primer encuentro oficial, que tuvo lugar en octubre de 2015, AlphaGo derrotó al tres veces campeón de Europa y segundo dan Fan Hui, saliendo victorioso de las cinco partidas de las que constaba el enfrentamiento.

Más tarde, en marzo de 2016, fue capaz de derrotar al 18 veces campeón del mundo y noveno dan Lee Sedol, considerado como el mejor jugador de la época. Por este hito le fue otorgado el noveno dan, máximo reconocimiento concedido a los jugadores profesionales. Netflix realizó un documental de dicho enfrentamiento llamado AlphaGo (Figura 2.17).

Por último, en enero de 2017, el equipo reveló que el jugador online "Master", el cual había conseguido una racha de 60 victorias consecutivas enfrentándose a jugadores profesionales, se trataba en realidad de una versión mejorada de AlphaGo denominada AlphaGo Master.


Figura 2.17: Imagen del documental AlphaGo [8] realizado por Netflix en el que se muestra el enfrentamiento entre AlphaGo y Lee Sedol. [6]

### AlphaGo Zero

En octubre de 2017 se publicó una versión mejorada de AlphaGo denominada AlphaGo Zero [7], que mostraba importantes diferencias con respecto a las versiones anteriores.

En primer lugar, se unieron ambas redes neuronales en una sola y se simplificó el método de búsqueda en árboles, de modo que ya no realizara simulaciones de los posibles resultados de la partida y basara sus decisiones únicamente en el estado de ésta.

En segundo lugar, se eliminó la primera fase del entrenamiento en la que AlphaGo entrenaba a partir de experiencias humanas, de modo que su aprendizaje no estaba condicionado por los límites del conocimiento humano.

Estos cambios y simplificaciones permitieron a AlphaGo Zero reducir drásticamente el tiempo de entrenamiento y superar rápidamente a sus predecesores. Tras sólo tres días de entrenamiento fue capaz de derrotar a la última versión de AlphaGo por 100 partidas a 0, y tras cuarenta días fue capaz de superar a AlphaGo Master. Por lo tanto, como dicen sus creadores, se puede afirmar sin temor a equivocarse que AlphaGo Zero es el mejor jugador de go de la historia.

### OpenAI Dota 2

OpenAI [22] es una compañía sin ánimo de lucro fundada por Elon Musk que tiene como objetivo promover y desarrollar el campo de la inteligencia artificial de modo que sea beneficioso y seguro

para toda la humanidad.

La compañía consideraría su objetivo cumplido incluso si son otros quienes lo lograsen, por lo que comparte de manera libre los resultados e información de todas sus investigaciones.

En 2016 hicieron público el desarrollo de un agente de inteligencia artificial basado en el Deep Reinforcement Learning capaz de derrotar a jugadores profesionales del videojuego Dota 2 [3] en enfrentamientos uno contra uno.

### Dota 2

Dota 2 (Figura 2.18) es un videojuego perteneciente al género MOBA. En este género, dos equipos formados por varios jugadores, cada uno de los cuales controla a un personaje con unas características propias, se enfrentan con el fin de destruir la base rival antes de que lo haga el enemigo.

Figura 2.18: Captura del videojuego Dota 2. [3]

En Dota 2, estos equipos están formados por cinco jugadores que se enfrentan en un mapa dividido en tres calles defendidas por varias estructuras denominadas "torres", y los jugadores cuentan con la ayuda de unidades no controlables que aparecen en dichas calles de manera periódica denominadas "creep".

Cada jugador controla a un personaje o "héroe" que cuenta con unos parámetros y habilidades únicas, los cuales puede reforzar mediante la obtención de experiencia y la compra de objetos. Tanto la experiencia como el oro necesario para realizar dichas compras provienen de la muerte de un componente del equipo rival, ya sea un héroe, un creep o una estructura.

Cuando el personaje controlado por un jugador muere, éste reaparece en la partida tras un tiempo, el cual depende directamente de su nivel y del valor de su equipamiento.

El primer equipo en derribar el edificio principal o "ancestro" de la base rival será el ganador, independientemente del nivel, el valor del equipamiento o el número de muertes obtenidas por el equipo rival.

Aparte de la modalidad tradicional existe una modalidad uno contra uno en la que dos jugadores se enfrentan en la calle central, saliendo victorioso aquel jugador capaz de derribar en primer lugar la torre rival o de obtener dos muertes del jugador enemigo.

Dota 2 es bastante conocido dentro de la comunidad de los videojuegos por su dificultad, por la dureza de su curva de aprendizaje y por ser uno de los juegos online más complejos.

#### Tecnología utilizada por OpenAI

La tecnología empleada por OpenAI es muy similar a la utilizada por AlphaGo, pero a diferencia del proyecto de DeepMind no utiliza ningún árbol de búsqueda.

Tal y como ocurre en el caso de AlphaGo Zero, el entrenamiento se basa principalmente en enfrentamientos del agente contra sí mismo. Además, también se entrenaron de manera exclusiva algunos aspectos del juego utilizando técnicas tradicionales de Reinforcement Learning, como por ejemplo el bloqueo de creeps (técnica que consiste en bloquear el camino de los creeps con el personaje de modo que se encuentren con los creeps del equipo rival más cerca la torre aliada, para que el jugador cuente con su protección).

La información recibida por el agente es prácticamente la misma que un humano podría obtener observando la pantalla, es decir, la posición de los elementos como héroes, creeps o estructuras que se encuentran dentro de su campo de visión. Sin embargo, también recibe alguna información extra, como su nivel de salud o el número de últimos golpes obtenidos (matar a un creep del equipo rival sólo proporciona oro si es el jugador quien asesta el último golpe).

### Resultados obtenidos

Tras obtener varias victorias durante su entrenamiento enfrentándose a jugadores amateur, el agente de OpenAI se enfrentó en varias ocasiones con jugadores profesionales.

El 7 de agosto de 2016 derrotó en varios enfrentamientos a tres partidas al exjugador profesional "Blitz" con un resultado de 3-0 y a los jugadores profesionales "Pajkatt" y "CC&C" con resultados de 2-1 y 3-0 respectivamente.

El 9 de agosto de 2016 derrotó al jugador número uno en el ranking mundial "Arteezy", con un resultado de 10-0.

El 10 de agosto de 2016 derrotó al mejor jugador del mundo en enfrentamientos uno contra uno "Sumail", con un resultado de 6-0.

El 11 de agosto de 2016 derrotó en un enfrentamiento al mejor de tres al excampeón del mundo y jugador más famoso del circuito profesional "Dendi", con un resultado de 2-0.

En la segunda publicación de OpenAI sobre este proyecto [26] pueden verse reflejados en vídeo algunos de los enfrentamientos que acabamos de mencionar.

### OpenAI Five

Tras haber conseguido unos resultados tan positivos en enfrentamientos uno contra uno, el equipo de OpenAI ha comenzado a desarrollar un nuevo agente denominado OpenAI Five [27], capaz de controlar a los cinco componentes de un equipo de manera simultánea.

Para ello, este nuevo agente cuenta con cinco redes neuronales, cada una de ellas destinada a controlar a uno de estos componentes.

La complejidad de controlar a un equipo completo es extremadamente superior a la de controlar a un único personaje en un enfrentamiento uno contra uno, ya que en un videojuego como Dota 2 la estrategia y la colaboración entre los miembros de un mismo equipo forman una parte imprescindible de los requisitos necesarios para conseguir la victoria.

Aunque OpenAI Five ya ha obtenido algunas victorias enfrentándose a equipos formados por jugadores humanos, en su primer enfrentamiento contra jugadores profesionales que tuvo lugar durante The International 2018 (nombre que recibe el mundial de

Dota 2) no obtuvo tan buenos resultados, ya que, aunque dejó algunos destellos de buen juego, acabó cayendo derrotado en ambas partidas.

No obstante, no se trataba de su versión definitiva, por lo que deberemos esperar a próximos eventos para comprobar si consigue superar estos resultados.

# Capítulo 3

# Descripción del problema y desarrollo

En este capítulo procederemos a explicar el desarrollo seguido por nuestro proyecto. Seguiremos el orden cronológico de los avances que realizamos en el proceso de investigación.

## 3.1. Lenguajes de programación y herramientas utilizadas

Decidimos utilizar Python [45] como lenguaje para desarrollar el proyecto frente a otros lenguajes como Java ya que es uno de los más utilizados en este área y porque sabíamos que podía utilizarse conjuntamente con la herramienta Keras para crear redes neuronales y también con los proyectos de OpenAI Gym, ya que ambas tecnologías se barajaron para ser parte del proyecto.

Como herramienta para crear redes neuronales barajamos tres opciones: Scikit-learn, Keras y TensorFlow, ordenadas de más alto a más bajo nivel. Teniendo en cuenta las necesidades de nuestro problema, decidimos que Keras era la que mejor se adaptaba a ellas. Por un lado, Scikit-learn no es lo suficientemente flexible y por otro, TensorFlow, tal como explican sus desarrolladores, sirve para realizar cálculos numéricos mediante diagramas de flujo de datos, que aunque se adapta muy bien a la creación de redes neuronales, siendo de hecho una de las herramientas más utilizadas para ello, es más genérico y añade una complejidad extra que hubiese complicado en exceso el desarrollo.

Keras 3.1 es un API de alto nivel para la creación de redes neuronales, que tiene a TensorFlow como back-end. La máxima de sus desarrolladores,

"*Ser capaces de pasar de las ideas a los resultados en la menor cantidad de tiempo*"

nos motivó a utilizarlo.


Figura 3.1: Logo de la herramienta Keras. [17]

Para cada variable de una red neuronal (función de pérdida, algoritmo de optimización, ...) existe en Keras una clase base cuyos métodos pueden ser sobrescritos a conveniencia. Podemos optar, por ejemplo, por usar una función de pérdida propia adaptada a nuestro problema, y utilizar un optimizador ya desarrollado. Keras nos permite con esta versatilidad centrarnos en diseñar de manera personalizada las partes que son exclusivas de nuestro problema, mientras utilizamos soluciones generales (que han demostrado ser efectivas) para las demás.

La mayoría de las funcionalidades que necesitamos para nuestro proyecto están implementadas en Keras (a modo de funciones de librería), lo que nos permite centrarnos en el diseño, pudiendo probar y evaluar distintas configuraciones simplemente cambiando parámetros. No obstante, todos los componentes que caracterizan a una red neuronal pueden ser reimplementados, siempre y cuando cumplan con las restricciones del API.

Utilizamos también algunas librerías de apoyo como NumPy y Matplotlib.


Figura 3.2: Logo de la librería Numpy. [40]

Según la página oficial de Numpy [40]:

"*NumPy es el paquete fundamental para programación científica en Python. Además de sus evidentes usos científicos, NumPy también puede ser utilizado como un eficiente contenedor*

*multidimensional de datos genéricos. Se pueden definir tipos arbitrarios. Esto permite a NumPy integrarse de manera rápida y sin problemas con una amplia variedad de bases de datos.*"

Contiene, entre otras cosas:

- Potentes arrays multidimensionales
- Funciones sofisticadas (de difusión)
- Herramientas para integrar código C/C++ y Fortran
- Álgebra lineal útil, la transformada de Fourier y capacidades para generar números aleatorios

La mayoría de las funciones ofrecidas por Keras tienen como entrada arrays de NumPy, por lo que su uso es obligatorio.

Para poder contrastar de manera objetiva los resultados de nuestros agentes, utilizamos Matplotlib [39], una librería para Python inspirada en MATLAB que permite crear gráficas bidimensionales. Lo usamos para generar gráficas en las que mostramos la evolución en las recompensas obtenidas por el agente a lo largo de los episodios de una ejecución, que nos ayuda a verificar si el agente aprende.

Como entornos de desarrollo utilizamos PyCharm, Jupyter, un editor de texto (Sublime Text) y la consola de comandos con Python3.

## 3.2. Descripción del problema

Para poner en práctica lo estudiado sobre cada uno de los campos el primer problema que nos planteamos fue el siguiente:

Un agente debe recorrer un tablero 8x8 (64 casillas o posiciones) desde una posición inicial hasta la salida. En su camino hacia la salida, tiene la opción de recoger una moneda posicionada en algún lugar indefinido del tablero y que le aportará una recompensa. Las acciones que puede tomar el agente son desplazarse a la izquierda, derecha, arriba o abajo (Figura 3.3).

Consideraremos resuelto el problema si se obtiene la solución óptima (recorrer el camino de longitud mínima hacia la salida que pase por la moneda) durante al menos 50 episodios consecutivos

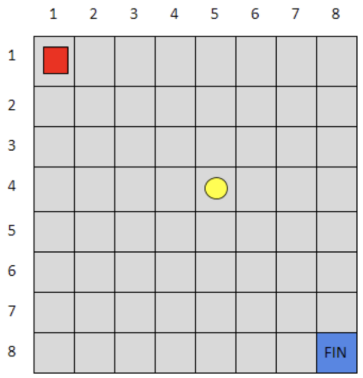


Figura 3.3: Representación gráfica del problema. El círculo amarillo representa al agente; los rectángulos rojo y azul la posición inicial y la salida respectivamente.

## 3.3. Solucion Q-Learning

La resolución usando Q-Learning se basa en el uso de la ecuación de Bellman que se explicó en la sección 2.2.5

$$Q(s_i, a_i) = (1 - \alpha) * Q(s_i, a_i) + \alpha * (r_i + \gamma * max_a Q(s_{i+1}, a))$$

que nos sirve para actualizar los valores-Q de cada entrada de la tabla-Q, método que utilizamos para representar la función-Q.

Es importante remarcar la diferencia entre cuando se está aplicando la regla de actualización y cuando se está consultando la tabla. En cada paso, la entrada de la tabla correspondiente al estado $s_i$, acción $a_i$ se modificará en función del valor que ya tenía, $Q(s_i, a_i)$, y del valor máximo que se puede obtener desde el estado siguiente, $Q(s_{i+1}, a)$. Estos dos valores se obtienen directamente realizando una consulta a la tabla-Q y no de manera recursiva. Si esto se hiciera de manera recursiva, en cada paso habría que aplicarla para todas las combinaciones estado-acción posibles, dando lugar a un bucle infinito.

Los valores dados a los hiperparámteros de nuestro agente son los siguientes:

Ratio de exploración. Su valor empieza siendo del 100 %, y se reduce cada vez que termina un episodio multiplicándolo por 0.9.

*Learning rate* ($\alpha$). Su valor es del 10 %.

Factor de descuento ($\gamma$). Su valor es del 90 %.

Hay varias formas de representar la existencia de la moneda y con cada una se obtienen diferentes resultados. Dos representaciones posibles son:

**No incluir la moneda en el estado**. El estado está constituido unicamente por las coordenadas x e y de la posición del agente. La primera vez (y solamente la primera vez) que pase por la posición de la moneda recibirá una recompensa.
Este planteamiento provoca que el agente intente regresar a la posición de la moneda aunque ya la haya cogido y no se le dé más recompensa. Como resultado, las entradas de la tabla-Q que lleven al agente a avanzar a esa casilla tendrán un valor muy alto. El agente no es capaz de discriminar si la moneda ya se ha cogido o no, por lo que repetirá esos pares estado-acción todo lo posible.
Por supuesto esta implementación no consigue un resultado óptimo y en muchas ejecuciones no logra terminar.

**Incluir la moneda en el estado**. El estado pasaría a estar formado por las coordenadas x e y del agente y un tercer valor que indica si se ha cogido la moneda o no. De esta forma se evitaría que el agente intentara volver a la posición de la moneda dado que las entradas de la tabla-Q de las posiciones adyacentes a la moneda son diferentes si se ha cogido la moneda o no. Dichas entradas tendrán un valor alto si no se ha cogido la moneda, mientras que si se ha cogido tendrán un valor bajo.
A pesar de aumentar el número de entradas de la tabla al doble es preferible esta implementación, ya que obtiene el resultado óptimo pasando por la moneda.

La última implementación demuestra que, modificando la representación del estado, el número de entradas de la tabla-Q varía exponencialmente, en este caso se duplica sólo por añadir una nueva entrada.

Esta implementación está disponible en GitHub en la URL:

`https://github.com/robrav01/TFG-RLDL/tree/master/Agente%20Reinforcement%20Learning`

### Resultados

Durante los primeros episodios los resultados son inestables, como se puede observar en la figura 3.4. El número total de movimientos para alcanzar la salida es muy elevado y en algunas ocasiones no

se encuentra la moneda. Esto se debe a que el agente empieza explorando (moviéndose aleatoriamente) un 100 % de las veces, reduciendo este nivel de exploración a un 90 % de su valor cada vez que termina un episodio. Además, en este momento los valores de la tabla-Q son aleatorios. Estos valores se irán modificando según el agente va aprendiendo.

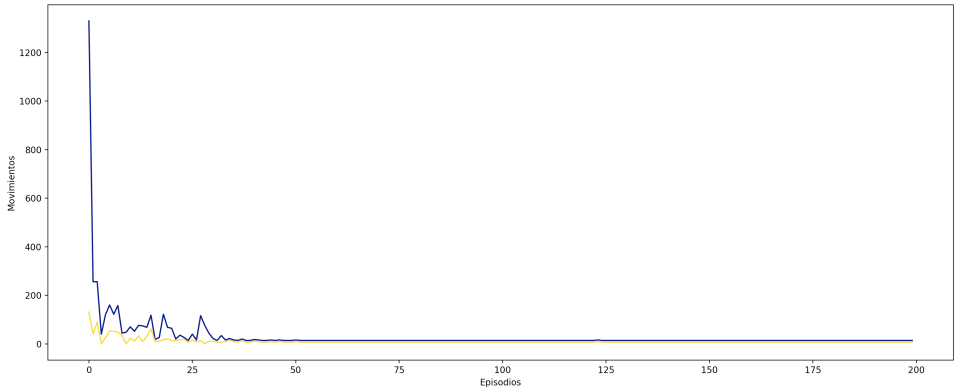


Figura 3.4: Movimientos que realiza en cada episodio el agente. En azul los movimientos que tarda en total en completar el recorrido y en amarillo los movimientos que ha tardado en coger la moneda.

A partir de cierto número de episodios el número de movimientos, tanto para terminar como para encontrar la moneda, converge. En torno al episodio 30, el número de movimientos se aproxima al óptimo (14 para terminar y 7 para encontrar la moneda). A partir del episodio 50, el nivel de exploración es mínimo y los resultados constantes.

### Limitaciones y conclusiones

Se puede observar que el Q-Learning es una técnica muy eficiente, por lo menos en problemas de esta escala, y que además ofrece muy buenos resultados, ya que en todos los casos consiguió resolver el problema obteniendo la mayor recompensa posible.

El mayor problema de la tabla-Q es que no es capaz de generalizar, de manera que el conocimento que ha adquirido para un estado no se puede aplicar en estados muy similares. Por lo tanto, es necesario realizar suficientes simulaciones como para visitar todos los estados posibles varias veces para calcular el valor de la función-Q para cada uno de ellos.

Además, el crecimiento exponencial del tamaño de la tabla-Q a medida que aumenta el número de entradas del estado hace que

sea cada vez más complicado calcular el valor de la función-Q para todas las entradas de la tabla.

## 3.4. Solución Deep Reinforcement Learning

Una vez entendido el funcionamiento del Q-Learning y conociendo sus limitaciones, pasamos a implementar este mismo problema pero utilizando como tecnología el Deep Reinforcement Learning. Como hemos explicado, nuestro objetivo será adaptar la solución Q-Learning sustituyendo la tabla-Q por una aproximación de la misma utilizando una red neuronal.

Como se explicó en la sección 2.4.1, el objetivo es reducir al mínimo la diferencia entre el valor aproximado por la red para un par estado-acción y su recompensa futura esperada, utilizando para ello una versión adaptada de la ecuación de Bellman:

$$min([Q(S_i, a_i)] - [r_i + \gamma * maxQ(S_{i+1}, a)])$$

A la hora de realizar la implementación Deep Reinforcement Learning se presentan varias cuestiones.

### Formato de la entrada y la salida

La primera de ellas puede ser qué formato tendrán la entrada y salida de la red neuronal. Para ello tuvimos en cuenta tres posibles propuestas:

- La entrada a la red neuronal es el estado y la función de activación de la última capa es SoftMax. Una capa que utiliza dicha función devuelve una distribución de probabilidad sobre el conjunto de posibles salidas (estas deben ser por tanto mutuamente excluyentes). Cada salida tendrá una probabilidad asociada (un valor entre 0 y 1)

  $$\sum_{i=1}^{4} p_i = 1$$

  En general, la función SoftMax en aprendizaje por refuerzo sirve para controlar la probabilidad de elegir una acción. En nuestro caso, pretendíamos utilizar dichas probabilidades como indicador del máximo valor de recompensa obtenible de

una acción en comparación con las demás. Acabamos descartando esta propuesta antes incluso de implementarla, ya que a la hora de entrenar a la red era imposible determinar cuál era el valor objetivo (es precisamente lo que intentábamos aprender).

- La entrada de la red neuronal es el par estado-acción y la salida el valor aproximado de la función-Q para dicho par, de manera que se realizaran los mínimos cambios posibles en el código. El objetivo sería que fuera similar a Q-Learning, salvo que las consultas a la tabla-Q se harían en su lugar a la red neuronal. Esta versión de la implementación está disponible en GitHub en la URL:
  `https://github.com/robrav01/TFG-RLDL/tree/master/Agente%20DRL/Version%201`
- El artículo del proyecto Playing Atari with Deep Reinforcement Learning [31] de la compañía DeepMind presenta un cambio en el formato de la entrada y salida de una red neuronal que resulta muy interesante. Como se explicó anteriormente, la entrada de la red neuronal está formada únicamente por el estado y la salida de la red por el valor aproximado de la función-Q para cada una de las posibles acciones. Esta opción de formato es perfectamente válida y ha dado buenos resultados en otros proyectos, ya que como se explicó en la sección 2.4.1 agiliza el proceso de entrenamiento al ser necesaria una única consulta para conocer todos los valores-Q asociados a un estado.

Las implementaciones finales de DRL de este proyecto utilizan la última opción.

Además, basándonos también en la propuesta de DeepMind, nos sumamos a la práctica habitual de acumular un conjunto de experiencias, tuplas de la forma (estado, acción, estado siguiente, recompensa, terminado), antes de aprender. A diferencia de ellos, nosotros no utilizamos *experience replay*, ya que no hay correlación entre estados consecutivos porque el agente se mueve de una casilla a otra instantáneamente.

### Topología de la red e hiperparámetros

Utilizamos una red neuronal de tres capas, con una capa oculta de 10 neuronas. Tanto la capa de entrada como la capa oculta utilizan la función de activación ReLU, mientras que la capa de salida utiliza la función de activación lineal. Ambas funciones de

activación se explicaron anteriormente en la sección 2.3.1.

Como función de pérdida utlizamos la MSE y como algoritmo de optimización el descenso de gradiente estocástico.

Los valores dados a los hiperparámteros de nuestro agente son los siguientes:

Ratio de exploración. Su valor empieza siendo del 100 %, y se reduce en un 1 % cada vez que termina una partida.

*Learning rate*. Su valor es de 0,05.

Factor de descuento ($\gamma$). Su valor es del 90 %.

### Método de inicialización

Otra cuestión relevante es la manera en que se inicializan los valores de la función-Q. En una tabla-Q, se pueden generar valores aleatorios acotados y almacenarlos en cada una de las entradas. En una red neuronal, los valores de la función-Q se obtienen en función de los valores de los pesos, por lo que sus valores iniciales dependen directamente del método de inicialización utilizado para los pesos.

La inicialización resultó ser un punto determinante en el proceso de aprendizaje. Los resultados obtenidos varían mucho según el método de inicialización utilizado. En nuestro caso, probamos varios métodos de inicializacion diferentes (de entre los ofrecidos por Keras) hasta dar con el adecuado (*glorot uniform* [43]) para nuestro problema, y no fue hasta entonces que el agente empezó a dar muestras de aprendizaje.

No contar con el método de inicialización más adecuado desde el primer momento retrasó el desarrollo e incluso nos hizo replantearnos el diseño de nuestro agente.

### Learning rate de la red neuronal

Un *learning rate* elevado puede suponer una aceleración del proceso de aprendizaje, pero también puede reducir la estabilidad del mismo. En nuestro caso, esto se traducía en un cambio brusco de los valores de la red provocando que la mejor acción para todos los estados fuera la misma.

Si, por ejemplo, la última acción antes de acabar (y que causa por tanto la obtención de la recompensa) es moverse a la derecha (r),

esta pasa a ser la mejor acción para todos los estados, salvo para aquellos estados en los que no es posible.

En la figura 3.5 se observa como la acción que lleva al estado final se propaga a todos los estados como la mejor acción, de forma que el agente aprende en bloque.

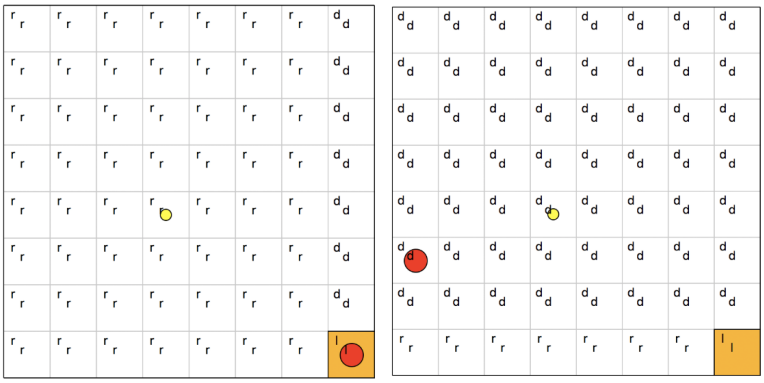

Figura 3.5: Los valores indicados en cada casilla representan la mejor acción posible utilizando sus iniciales en inglés (u-up, d-down, l-left, r-right). El valor que se encuentra más arriba indica la mejor acción si aún no se ha recogido la moneda y el que se encuentra más abajo indica la mejor acción una vez se ha recogido.

Otra situación que se podría dar es que no se minimizara la función de pérdida, que se traduce en nunca llegar a un objetivo. Por tanto, redujimos su valor de 0.1 a 0.05.

### Cambiar la codificación del estado a *One-hot*

Con el objetivo de que los estados fueran más diferentes entre sí y que la red neuronal fuera capaz de distinguir mejor estados cercanos, cambiamos la codificación de nuestro estado por una codificación *One-hot*.

La codificación *One-hot* es un método de representación en el que $n$ valores son representados por un vector de tamaño $n$. Todas las posiciones de dicho vector tienen valor 0, excepto una que tiene valor 1. La posición del valor 1 dentro de dicho vector será diferente para cada uno de los $n$ valores.

Un ejemplo de codificación *One-hot* sería codificar los colores rojo, amarillo y azul como `100` (*rojo*), `010` (*amarillo*) y `001` (azul).

La diferencia entre dos valores representados por $n$ (para $n$ = cardinalidad del conjunto de posibles valores) bits en los que únicamente uno está activado al valor 1 es mayor que la diferencia

entre dos valores que estén entre 0 y *n-1*.

En nuestro caso, utilizamos *One-hot* para codificar las coordenadas del agente dentro del tablero, haciendo que la entrada de la red estuviera formada por 8 bits para la posición $x$, 8 bits para la posición $y$ y un bit para indicar si la moneda había sido recogida o no.

El estado [2,6,0] en codificación one-hot sería:

$$[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$$

### Recompensas significativas

El valor de las recompensas debe ser significativo, de modo que obtener una suponga una repercusión para el valor de la red, tanto para el estado desde el que se consiguió como para sus estados adyacentes. En nuestro caso el valor de las recompensas dependía del número de movimientos y su valor era tan bajo que prácticamente no tenía impacto, por lo que cambiamos su valor por uno constante.

### Reducción más rápida de la exploración

En la sección 2.2.4 hemos visto que inicialmente el agente explora el espacio de estados-acciones y a medida que descubre qué pares le acercan más a la solución comienza a explotar ese conocimiento. La velocidad con la que desciende la exploración, o lo que es lo mismo, con la que se incrementa la explotación, es un factor muy a tener en cuenta en el proceso de aprendizaje.

Con la configuración de exploración-explotación que teníamos en un primer momento, aunque el agente ya hubiera encontrado la solución óptima del problema, no podía apreciarse una mejora en los resultados ya que el ratio de exploración era aún demasiado elevado (mayor del 70 %, lo que se traduce en que 7 de cada 10 acciones tomadas por el agente son aleatorias).

Decidimos reducir su valor más rápidamente, pero sólo para aquellos episodios en los que el agente obtenga recompensa. Nuestra idea era que para el momento en el que el agente conociera la solución óptima el ratio de exploración estuviera en torno al 20 %. Como veremos en el apartado de resultados, esto propiciará que los resultados se estabilicen antes en el óptimo.

Esta versión de la implementación está disponible en GitHub en la URL:

`https://github.com/robrav01/TFG-RLDL/tree/master/Agente%20DRL/Version%204`

## Resultados

Aplicando todo lo mencionado a nuestra implementación conseguimos que el funcionamiento fuera el esperado no sólo para el problema que nos ocupaba sino también para varias modificaciones del mismo, lo que demostró que nuestro método de aprendizaje era correcto.

Algunos de los problemas con los que lo probamos fueron obligar al agente a recoger la moneda para poder terminar la partida o modificar la posición de la salida (Figura 3.6), de modo que para resolver el problema no tuviera que aprender únicamente a ir abajo y a la derecha sino que tuviera que bajar a recoger la moneda y después subir de nuevo para terminar.

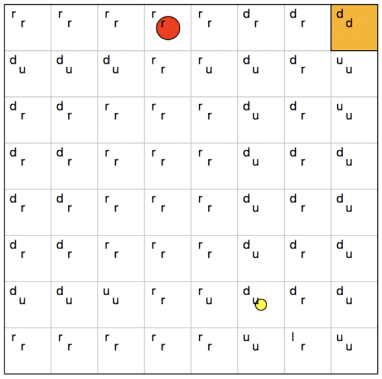

Figura 3.6: Se observa como incluso complicando el problema (ahora tiene que bajar a por la moneda y subir hacia la salida) es capaz de obtener la máxima puntuación. Además, se aprecia cómo incluir la moneda en el estado ayuda a su resolución.

Los resultados obtenidos (Figura 3.7) son similares a los obtenidos con la implementación Q-Learning. Se muestra inestable en los primeros episodios (debido a los altos niveles de exploración) y va mejorando con la sucesión de los mismo. Nuevamente, en torno al episodio 50 los resultados se estabilizan en el óptimo.

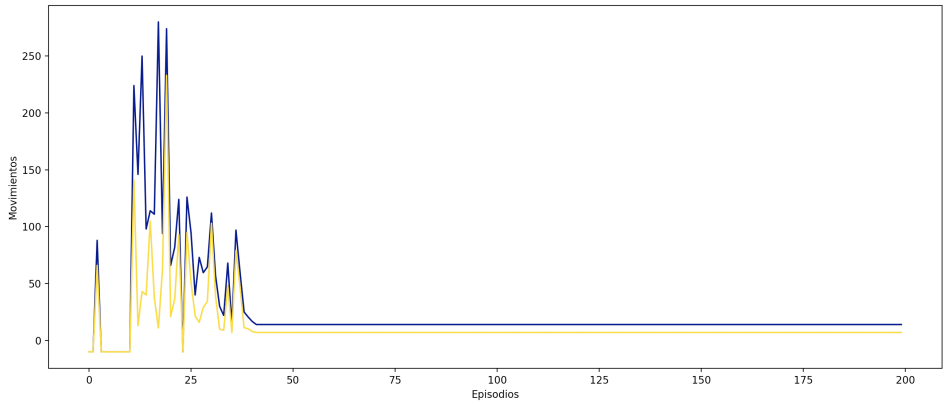


Figura 3.7: Media de movimientos que tarda en el agente alcanzar la salida (en azul) y obtener la moneda (en amarillo) en cada episodio (300 movimientos). Los puntos con valores negativos son episodios en los que no se ha llegado a alcanzar la salida.

### Limitaciones y conclusiones

Pudimos observar que los resultados obtenidos por el Deep Reinforcement Learning son tan positivos como los obtenidos por el Q-Learning, ya que en todos los casos fue capaz de resolver el problema utilizando la solución óptima.

En lo que a eficiencia se refiere, los resultados son algo peores en comparación con el Q-Learning, aunque también es cierto que, debido a los problemas con los que nos encontramos durante el desarrollo, no pudimos aplicar ninguna de las mejoras del proceso de aprendizaje propuestas por DeepMind, como el uso de una *target network*.

Como parte negativa, la complejidad de las redes neuronales es mucho mayor que la de la tabla-Q. Para problemas simples no merecería la pena su uso, pero cuando estos escalan es obligatorio.

Los parámetros utilizados para crear la red neuronal, como la tasa de aprendizaje, la función de inicialización, etc., cambian drásticamente los resultados obtenidos. Esto resulta un problema porque en muchos casos su inicialización es cuestión de ensayo y error y ralentiza el desarrollo del proyecto.

# Capítulo 4

# OpenAI

Según la página oficial de OpenAI Gym [28]:

> "*Gym es un conjunto de herramientas para el desarrollo y la comparación de algoritmos de Reinforcement Learning. No realiza ninguna suposición sobre la estructura de tu agente, y es compatible con cualquier librería de computación numérica, como TensorFlow o Theano. La librería de Gym es una colección de problemas de testeo - entornos - que puedes usar para entrenar tus algoritmos de reinforcement learning. Estos entornos tienen una interfaz común, permitiéndote escribir algoritmos genéricos.*"

Esta interfaz común a la que se refiere la descripción proporciona a los desarrolladores que quieran utilizarla el estado del problema, valor de la recompensa para la última acción y si el problema ha terminado o no.

Debido a la facilidad para combinar esta herramienta con nuestro agente decidimos utilizarla y resolver alguno de los problemas que propone, ya que se trata de un sistema de pruebas conocido internacionalmente que nos permitiría poner a prueba la corrección de la implementación de nuestro agente.

A continuación describiremos dichos problemas y expondremos los resultados obtenidos y las modificaciones aplicadas a nuestro agente.

## 4.1. CartPole

Este problema consiste en un pequeño carro que debe sostener sobre él un poste al que está unido por un único punto (Figura 4.1). Moviendo el carro de manera horizontal hacia la izquierda o la derecha, el agente deberá corregir la posición del poste para que éste no caiga por debajo de la posición del carro.

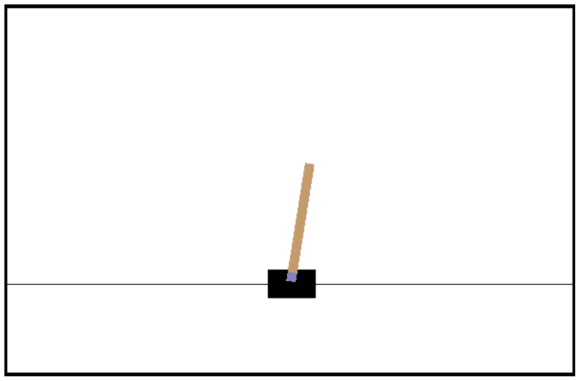

Figura 4.1: Captura de nuestro proyecto ejecutando el problema CartPole. [29]

El estado está compuesto por la posición del carro, su velocidad, el ángulo del poste y la velocidad de la punta del poste. Las acciones disponibles son dos: moverse a la izquierda o moverse a la derecha. El estado inicial para todas las ejecuciones del problema será aletorio, pero limitado por unos valores determinados para evitar que la partida empiece en un estado en el que el poste sea imposible de estabilizar.

La recompensa tiene valor 1 para cada paso dado, incluído el último.

El código original de este problema se puede comprobar en el siguiente enlace:

`https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`

### Modificaciones realizadas

Para su resolución no fue necesario realizar más modificaciones que las necesarias para adaptar nuestro agente a las dimensiones del problema, es decir, establecer las acciones disponibles a moverse a la izquierda o a la derecha, reducir el número de salidas

de la red neuronal a dos para que coincidiera con el número de acciones y reducir el número de entradas de la red a cuatro para que coincidiera con el formato del estado.

Además, se añadió el uso de *experience replay* para optimizar el proceso de aprendizaje.

### Topología de la red e hiperparámetros

Utilizamos una red neuronal de tres capas, con una capa oculta de 30 neuronas. Tanto la capa de entrada como la capa oculta utilizan la función de activación ReLU, mientras que la capa de salida utiliza la función de activación lineal.

Como función de pérdida utlizamos la MSE y como algoritmo de optimización Adam [44].

Los valores dados a los hiperparámteros de nuestro agente son los siguientes:

Ratio de exploración. Su valor empieza siendo del 100 %, y se reduce cada vez que termina un episodio multiplicándolo por 0.999.

*Learning rate*. Su valor es del 0,001.

Factor de descuento ($\gamma$). Su valor es del 95 %.

### Resultados obtenidos

El rendimiento ofrecido por nuestro agente fue satisfactorio, ya que se puede observar que aprende de una manera adecuada en la mayoría de las ejecuciones, logrando incluso buenos resultados en algunas de ellas. Esto se puede observar en la figura 4.2.

No obstante, comparando nuestros resultados con los obtenidos por otros desarrolladores, llegamos a la conclusión de que mejorando la estabilidad del proceso de aprendizaje y encontrando una combinación de hiperparámetros más acertada hubiésemos conseguido que éstos fueran aún mejores.

Nuestra implementación de CartPole está disponible en GitHub en la URL:

`https://github.com/robrav01/TFG-RLDL/tree/master/CartPole`

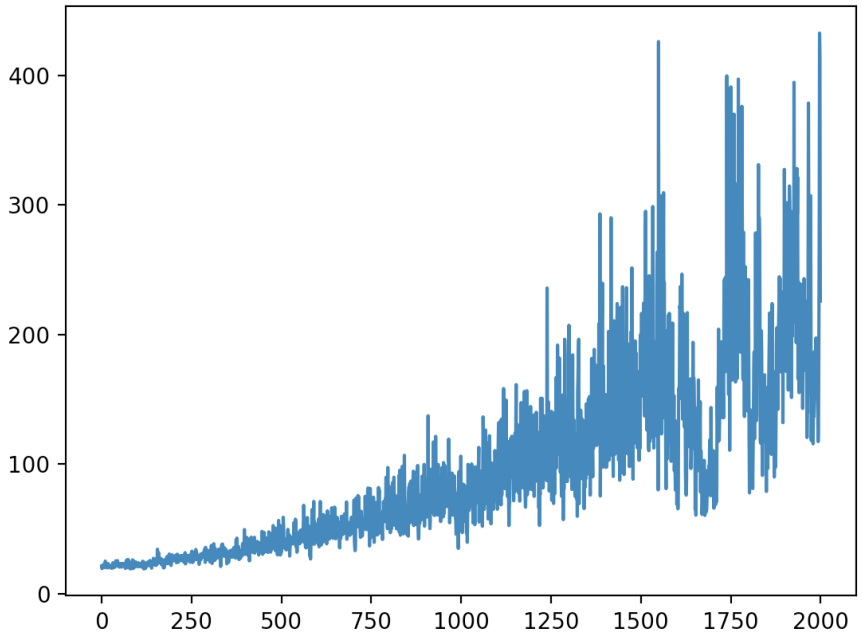


Figura 4.2: Media de movimientos tomados por el agente antes de que caiga el poste (eje y) en cada episodio de aprendizaje (eje x).

## 4.2. Taxi

En este problema, un bloque que actúa en representación de un taxi deberá recorrer un tablero en el que hay marcadas cuatro posiciones. El agente deberá controlar dicho taxi, recogiendo pasajeros de alguna de las localizaciones marcadas y llevándolos a su destino con la mayor rapidez posible (Figura 4.3).

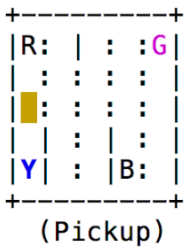


Figura 4.3: Captura de nuestro proyecto ejecutando el problema Taxi [30] de OpenAI.

El estado está compuesto por un único número que se obtiene combinando mediante varias operaciones la fila y la columna en

las que se encuentra el taxi, la posición del pasajero y la posición del destino.

Las acciones disponibles son seis: moverse hacia arriba, hacia abajo, a la izquierda, a la derecha, recoger a un pasajero y dejar a un pasajero.

La recompensa tiene valor 20 en el momento en que se deja al pasajero en su destino y de -1 para cualquier otro paso. Además, si se intenta recoger o dejar a un pasajero en un punto incorrecto la recompensa es de -10.

El código original de este problema se puede comprobar en el siguiente enlace:

`https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py`

### Modificaciones realizadas

Los principales cambios consistieron en adaptar el agente a las dimensiones del problema, estableciendo las acciones disponibles a las mencionadas anteriormente y adaptando el número de entradas y salidas de la red para que coincidieran con el formato del estado y el número de acciones respectivamente.

Por otro lado, en este problema sólo se obtienen recompensas positivas al alcanzar el objetivo (dejar al pasajero en su destino), por lo que es importante que el conjunto de entrenamiento cuente con suficientes ejemplos exitosos. Para ello, probamos dos alternativas:

1. Utilizar *experience replay* y aumentar el número de pasos (acciones tomadas por el agente) que componen un episodio para así incrementar las probabilidades de que el conjunto de entrenamiento contenga experiencias exitosas.
   En la figura 4.4 se observa la curva de aprendizaje del agente en una de las mejores ejecuciones. En este caso, el aprendizaje está fuertemente condicionado por la aparición de experiencias exitosas en los conjuntos de entrenamiento de cada episodio, por lo que no está garantizado que obtengamos buenos resultados.
2. Hacer que cada conjunto de entrenamiento este compuesto por todas las experiencias de una única partida completa, esto es, que el agente almacene experiencias hasta llegar a un

estado terminal (en este caso, llevar a todos los pasajeros a su destino).
De esta manera, se garantiza que en el conjunto de entrenamiento haya experiencias positivas, de forma que cada vez que entrena el agente aprenda una nueva forma de resolver el problema (o refuerce una ya conocida). Esto garantiza un aprendizaje constante. Los resultados se resumen en la figura 4.5.

Debido a la mayor estabilidad de los resultados a lo largo de varias ejecuciones, acabamos decantándonos por la segunda opción.

### Topología de la red e hiperparámetros

Utilizamos una red neuronal de tres capas, con una capa oculta de 15 neuronas. Tanto la capa de entrada como la capa oculta utilizan la función de activación ReLU, mientras que la capa de salida utiliza la función de activación lineal.

Como función de pérdida utlizamos la MSE y como algoritmo de optimización Adam [44].

Los valores dados a los hiperparámteros de nuestro agente son los siguientes:

Ratio de exploración. Su valor empieza siendo del 100 %, y se reduce cada vez que termina un episodio multiplicándolo por 0.9994.
*Learning rate*. Su valor es del 0,001.
Factor de descuento ($\gamma$). Su valor es del 95 %.

### Resultados obtenidos

Los resultados obtenidos fueron muy positivos, ya que logramos que el agente mostrara un aprendizaje estable y constante y que consiguiera niveles de recompensa prácticamente óptimos.

No obstante, se trata de un problema de representación sencilla. La tabla-Q para este problema contendría 3000 entradas, correspondientes a la combinación de los 500 posibles estados con las 6 acciones disponibles en cada uno de ellos, por lo que podría haberse resuelto mediante Q-Learning de una manera más eficiente.

Además, al estar el estado representado por un único número, la red neuronal no puede generalizar ya que estados similares se representan con números totalmente diferentes.

Nuestra implementación de Taxi está disponible en GitHub en la URL:

`https://github.com/robrav01/TFG-RLDL/tree/master/Taxi`

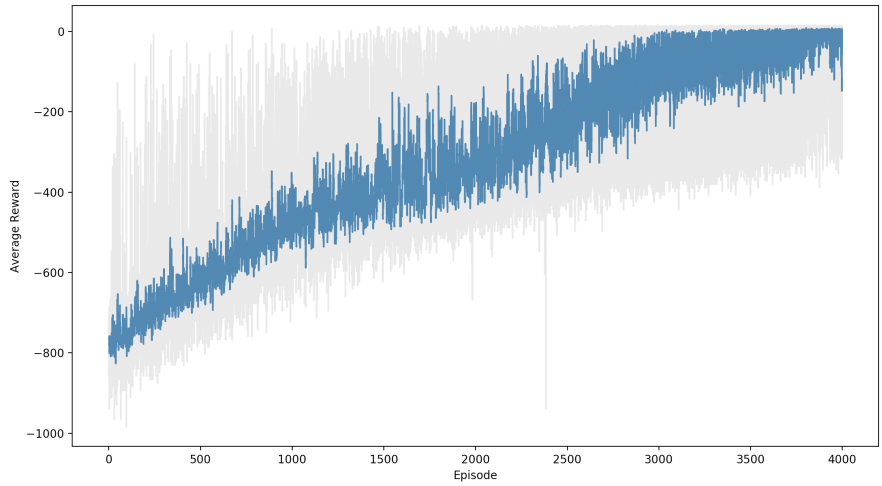


Figura 4.4: Recompensas total (en gris) y promedio (en azul) obtenidas por el agente con experience replay en cada episodio (1000 pasos). Se estabiliza cerca del óptimo a partir de los 3000 episodios.

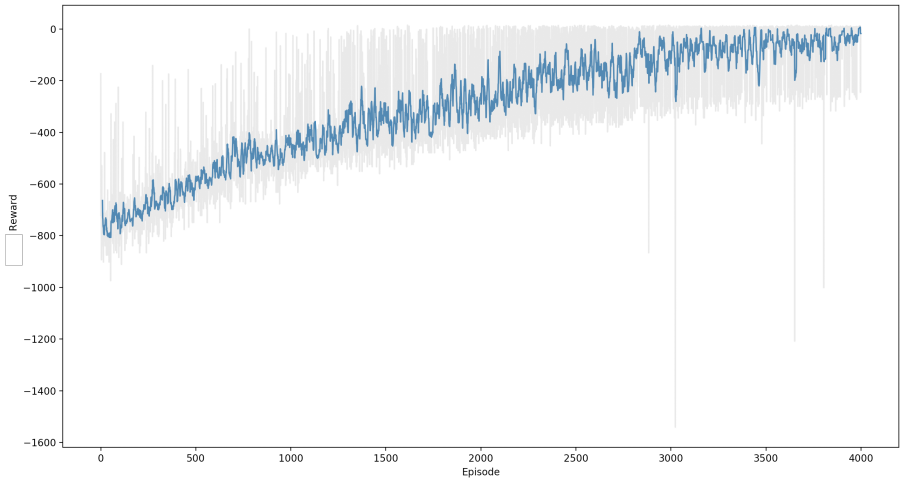


Figura 4.5: Recompensas total (en gris) y promedio (en azul) obtenidas por el agente utilizando para el entrenamiento todas las experiencias almacenadas en cada episodio. Se acerca al óptimo por primera vez en el episodio 2800.

# Capítulo 5

# Conclusiones

El Deep Reinforcement Learning es una tecnología muy poderosa capaz de resolver problemas de gran complejidad de manera eficiente y obteniendo muy buenos resultados.

Al ser una técnica de aprendizaje por refuerzo, basa su proceso de aprendizaje en determinar la calidad de las acciones posibles para cada estado. Esto permite resolver problemas para los que no se conoce cuál es la solución óptima pero sí si una vez tomada una acción ésta ha sido positiva o negativa, algo que no pueden hacer otros métodos de aprendizaje automático como por ejemplo los basados en aprendizaje supervisado.

Hemos visto que el Q-Learning es una buena técnica para resolver problemas sencillos mediante aprendizaje por refuerzo. Al combinar esta técnica con el uso de redes neuronales, el Deep Reinforcement Learning presenta la ventaja de poder resolver problemas de una escala mayor.

Como ejemplo tenemos el CartPole. El estado se representa con valores continuos, que son una entrada válida a una red neuronal pero que son inapropiados para una representación tabular de la función-Q. Existen ejemplos mucho más representativos de esta capacidad que ofrece el DRL sobre el Q-Learning, como los proyectos de DeepMind y OpenAI comentados en la sección 2.4.4.

Sin embargo, el nivel de complejidad que añade al proceso de aprendizaje hace que el uso del Deep Reinforcement Learning no sea recomendable para problemas de pequeña escala, siendo mucho más sencilla su resolución mediante Q-Learning.

Parte de esta complejidad se debe a que dicho proceso está fuertemente condicionado por la configuración de los parámetros pro-

pios de la red neuronal (función de *activación*, algoritmo de *optimización*, función de *pérdida*, número de neuronas por capa, ...), lo que dificulta la resolución y optimización del problema. Esto es debido a que el valor óptimo de muchos de ellos solamente se puede obtener mediante el método de ensayo y error. Pudimos comprobarlo por nosotros mismos al implementar nuestro agente. Nos llevó mucho tiempo observar aprendizaje y tuvimos que realizar diversos cambios, algunos de los cuales no eran necesarios para arreglarlo, porque nuestra función de inicialización no era la adecuada y nuestro *learning rate* tenía un valor demasiado elevado para que los resultados llegaran a la solución óptima.

A esta complejidad se suman los elevados costes que esta técnica presenta en tiempo y recursos hardware, que hacen que no esté al alcance de todo el mundo.

No obstante, las redes neuronales se adaptan bien al uso de GPUs para paralelizar el proceso de aprendizaje, de modo que el agente sea entrenado en múltiples entornos de manera simultánea. Esto permite reducir drásticamente el tiempo de entrenamiento y hacer mucho más eficiente el proceso de aprendizaje debido a la mayor variedad de las muestras.

Pese a todo, los resultados ofrecidos por el Deep Reinforcement Learning y la evolución que ha demostrado resolviendo problemas cada vez más complejos desde su aparición en el año 2013 demuestran que se trata de una tecnología de presente y de futuro que es muy posible que se convierta, si no lo es ya, en el área más influyente dentro del campo de la inteligencia artificial.

## 5.1. Trabajo Futuro

A partir del estado actual del proyecto, se podría aplicar lo aprendido hasta el momento para resolver problemas de mayor complejidad.

Un primer paso sería utilizar un agente similar en problemas más complejos en un entorno sencillo y ya conocido, como por ejemplo los juegos de Atari ofrecidos por OpenAI Gym.

Posteriormente, tratar de mejorar su funcionamiento utilizando algunas de las técnicas descritas en el capítulo 2, como por ejemplo el uso de una target network, o incluso aplicar algunas de las evoluciones de las DQN explicadas en ese mismo capítulo, como las DRQN o las A3C.

Como último paso, sería interesante centrarse en uno de los problemas resueltos y tratar de optimizar los resultados obtenidos.

# Conclusions

Deep Reinforcement Learning is a very powerful technology able to solve complex problems efficiently and obtaining very good results.

Its learning process is based in determinate the quality of the possible actions for each state, so its allows to solve problems which we dont know the optimal solution but we know if taking a determinate action has a positive or negative result. That is something other Machine Learning methods cannot do, for example the ones based on Supervised Learning.

We have seen that Q-Learning is a good technique for solving simple problems through Reinforcement Learning. By combining this technique with the use of neural networks, Deep Reinforcement Learning has the advantage of being able to solve problems of a larger scale.

As an example we have CartPole. The state is represented with continuous values, but they are inappropriate for a Q-table representation. There are much more representative examples of this capability shown by DRL above Q-Learning, as DeepMind and OpenAI projects commented in section 2.4.4.

However, the level of complexity that it adds to the learning process makes the use of Deep Reinforcement Learning not recommendable for small scale problems, being Q-Learning solutions much simpler.

Part of this complexity is due to the the fact that the learning process is strongly conditioned by the configuration of parameters of the neural network (activation function, optimization algorithm, loss function, neurons per layer, ...), which difficulties the solution and optimization of the problem. This is because the optimal value of many of them can only be obtained by the trial and error method. We could check this by ourselves while implementing our agent. It took us a long time to observe learning and we had to

made several changes, some of which were unnecessary for fixing the problems, because our initialization function was not the right one and our learning rate was too high for the results to reach the optimal solution.

To this complexity we have to add the high cost that this method has in time and hardware resources. This makes that not everyone can afford DRL.

Nevertheless, neural networks suit well with the use of GPUs to parallelize the learning process, so the agent can be trained in multiple environments simultaneously. This allows to reduce the training time drastically and make the learning process much more efficient due to the higher variety of the samples.

Despite all of this, Deep Learning obtained results and the evolution that it has shown solving problems more complex each time since its origin in 2013 prove it is a present and future technology that it is likely to become, if it is not already, the most influential subject within AI area.

## Future Work

From the current state of the project, the knowledge learnt may be applied to solve more complex problems.

A first step would be using a similar agent in more complex problems in a simple and well-known environment, such as the Atari games offered by OpenAI Gym.

Later, try to improve its performance by using some of the techniques described in chapter 2, such as the use of a target network, or even by applying some of the evolutions of the DQN explained in the same chapter, as the DRQN or the A3C.

As a last step, could be interesting to focus in one of the solved problems and try to optimize the obtained results.

# Capítulo 6

# Aportación de los participantes

Al tratarse de un proyecto de investigación sobre una materia que ninguno de los autores conocía más allá de sus conceptos más básicos la mayoría del trabajo se realizó de manera conjunta.

Para asentar las nociones básicas, decidimos cada uno investigar por separado cada uno de los aspectos relevantes de la materia, para posteriormente hacer una puesta en común de los mismos, debatir sobre aquellos puntos en lo que no estábamos de acuerdo e intentar resolver las dudas que pudiéramos tener con el fin de obtener una base sólida sobre la que empezar nuestro proyecto.

A la larga, mediante este proceso de debate, llegamos a mejorar el aprendizaje y resolvimos las dudas y problemas que se nos iban planteando, no sólo a la hora de tener el suficiente conocimiento teórico si no también en el desarrollo de código.

Con respecto a la parte práctica, y sin dejar en ningún momento de investigar para reforzar nuestro conocimiento teórico, empezamos tratando de resolver de manera individual un problema común siguiendo el modelo que acabamos de mencionar: el problema del tablero con la moneda utilizando Q-Learning.

Como mencionamos en el apartado de desarrollo, nuestros principales problemas fueron la interpretación de la ecuación de Bellman y su aplicación en la práctica y la representación del estado del problema (la inclusión o no de la moneda en el mismo), que acabamos resolviendo conjuntamente.

Nuestro siguiente paso fue ampliar nuestro conocimiento sobre redes neuronales más allá de lo aprendido en el grado. Decidimos

hacerlo de manera conjunta desde un principio, ya que considerábamos que era un tema complicado y de gran importancia para el desarrollo y no queríamos que ninguno de los participantes se quedara atrás.

Comenzamos a resolver con el uso de la herramienta scikit-learn los problemas más sencillos de clasificación, pero tras sopesarlo y teniendo en cuenta las necesidades de nuestro proyecto, decidimos sustituirla por la herramienta Keras debido a su mayor flexibilidad.

A partir de este momento, todo lo referente al desarrollo de nuestro agente DRL para la resolución del problema del tablero también se realizó colaborativamente, debido principalmente a la dificultad para modular el código y a la necesidad de enfrentarnos juntos a los problemas con los que nos topamos, que eran muchos y en su mayoría estructurales. Tras mucho tiempo y esfuerzo, logramos conseguirlo.

Con nuestro agente como base, la resolución de los problemas de OpenAI Gym se reducía a adaptarlo a cada problema y encontrar la mejor configuración de los hiperparámetros para cada uno de ellos. Por ello, cada uno de los participantes trató de resolverlos por su cuenta para posteriormente utilizar en el proyecto aquella solución que ofrecía mejores resultados.

La redacción de la memoria se hizo de manera conjunta para todos los capítulos a excepción del segundo, de modo que cada uno de los participantes explicara una de las áreas de la inteligencia artificial que hemos aplicado en nuestro proyecto.

## 6.1. David Alba Corral

Tras decidir qué temas principales queríamos tratar, me correspondió presentar el Reinforcement Learning. De los tres temas (RL, DL y DRL), éste es el más antiguo y por tanto del que más bibliografía hay disponible. Me apoyé principalmente en libros que tratasen sobre Deep Reinforcement Learning, porque todos ellos cuentan con un capítulo introductorio sobre RL, que es justo lo que yo necesitaba: presentar las bases de nuestro proyecto para facilitar el trabajo de mis compañeros en la redacción de los posteriores capítulos.

A pesar de que es un tema muy activo dentro de la inteligencia artificial y la bibliografía al respecto es amplia, me resultó difícil

de explicar, especialmente mantener un equilibrio entre la formalidad, más precisa pero menos intuitiva, y el uso de metáforas y analogías, que facilitan transmitir las ideas al lector pero con las que se puede llegar a perder practicidad.

Además, como los tres temas tratados están relacionados entre sí y tienen puntos comunes, tuve (en realidad, tuvimos) que tener especial cuidado en presentar todos aquellos conceptos que mis compañeros necesitaran dar por hecho en sus respectivos capítulos.

## 6.2. Rodrigo Bravo Antón

El tema que tuve que tratar fue el Deep Learning.

Dado que es básico para nuestro desarrollo entender los fundamentos de esta rama de la inteligencia artificial, quería que mi parte sirviese como toma de contacto para entender su funcionamiento.

Al ser un tema que se encuentra en el foco de atención desde hace unos años, encontré una cantidad de información abrumadora. Con tanta información era difícil seleccionar qué debía formar parte del apartado.

Barajé distintos apartados que debía añadir y muchos de ellos no están incluidos. Una de las razones fue que me parecía que estaba profundizando demasiado o que estaba dando información que al final podía no ser relevante para nuestro proyecto. Al fin y al cabo es un proyecto de Deep Reinforcement Learning, y aunque queremos que los lectores que no conozcan Deep Learning se hagan una idea de su funcionamiento, es un tema demasiado extenso y recomiendo que se utilicen nuestras referencias para adquirir un conocimiento más avanzado.

Opté por basar la explicación del modelo en el curso Machine Learning Crash Course de Google porque explica muy bien los conceptos básicos y en su momento me sirvió para mi propio aprendizaje.

Finalmente, los dos ejemplos de computer vision que utilicé son intencionales, dado que utilizan una tecnología muy similar a la que se usa para crear agentes que jueguen automáticamente y son proyectos punteros, y son sólo dos dado que los ejemplos más destacados debían estar en DRL.

## 6.3. Javier García Rodriguez

Por mi parte, estuve encargado de explicar el Deep Reinforcement Learning.

Aunque había muchas fuentes referentes al tema, fue una tarea algo más complicada de lo que me esperaba. La mayoría de estas fuentes ofrecía una explicación detallada del funcionamiento del Reinforcement Learning y del Q-Learning, pero no entraba en demasiados detalles a la hora de explicar cómo dar el paso hacia el Deep Reinforcement Learning sustituyendo la tabla-Q por una red neuronal como herramienta para aproximar el valor de la función-Q.

Finalmente, utilicé como referencias la publicación del equipo de DeepMind sobre su proyecto Playing Atari with Deep Reinforcement Learning [31] y el libro *Fundamentals of Deep Learning* [12], que en su capítulo final describe en mayor profundidad este tema basándose en dicha publicación.

Ambas fuentes, además de explicar la unión del Q-Learning con el uso de redes neuronales para crear las Deep Q-Networks o DQN, explican algunas de las mejoras aplicadas por el equipo de DeepMind para mejorar la eficiencia y la estabilidad del proceso de aprendizaje, como el uso del *experience replay* o de una *target network*.

Además, en el mencionado capítulo del libro *Fundamentals of Deep Learning* [12] se describen algunas de las versiones posteriores de las DQN desarrolladas también por DeepMind. Me pareció muy interesante añadirlo, ya que explican varias mejoras utilizadas tanto para reducir los tiempos de entrenamiento como para mejorar los resultados obtenidos y son mucho más próximas a las versiones más punteras de esta tecnología utilizadas hoy en día.

Para finalizar con este apartado, me pareció acertado explicar algunos de los proyectos más destacados que utilizan como base el Deep Reinforcement Learning, y terminé decantándome por AlphaGo y OpenAI Dota 2.

La elección de AlphaGo estuvo debida a que es un proyecto posterior del equipo de DeepMind, lo cual nos permitiría ver la evolución de la tecnología, y sobre todo a que se trata del proyecto que dio su verdadera fama al Deep Reinforcement Learning y lo puso a la vista de todo el mundo, incluso de aquellas personas que no forman parte del campo de la informática.

Por su parte, la elección de OpenAI Dota 2 se debió a que obtuvo muy buenos resultados en el año 2016 enfrentándose a jugadores profesionales de un videojuego tan complicado como Dota 2 y a que es un proyecto que sigue activo a día de hoy, pudiendo comprobar sus últimos avances durante el mes de agosto de este mismo año, 2018.

# Bibliografía


[1] L.Beam, Andrew. *Deep Learning 101 - Part 1: History and Background*. 2017.

[2] Berkeley AI Materials, `http://ai.berkeley.edu/project_overview.html`.

[3] Dota 2, `es.dota2.com/`.

[4] *Why Deep Learning over Traditional Machine Learning?*. Towards Data Science, 2018, `https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063`.

[5] DeepMind, `deepmind.com/research/alphago/`.

[6] *AlphaGo Movie*. AlphaGo Movie, `www.alphagomovie.com/`.

[7] *AlphaGo Zero: Learning from Scratch*. DeepMind, `deepmind.com/blog/alphago-zero-learning-scratch/`.

[8] *AlphaGo | Netflix*. Netflix Official Site, 2018, `www.netflix.com/es/title/80190844`.

[9] *Amazon Rekognition Videos e Imágenes AWS*. Amazon, `aws.amazon.com/es/rekognition/`.

[10] *A Beginner's Guide to Neural Networks and Deep Learning*. Deeplearning4j, `deeplearning4j.org/neuralnet-overview`.

[11] *A Beginner's Guide to Neural Networks and Deep Learning*. Skymind, `skymind.ai/wiki/neural-network`.

[12] Buduma, Nikhil, and Nicholas Locascio, *Fundamentals of Deep Learning: Designing next-Generation Machine Intelligence Algorithms*. O'Reilly Media, 2017.

[13] Dutta, Sayon. *Reinforcement Learning with TensorFlow*. Packt Publishing Limited, 2018.

[14] Sorta Insightful, *Deep Reinforcement Learning Doesn't Work Yet*. www.alexirpan.com/2018/02/14/rl-hard.html.

[15] *Go*. Wikipedia, 2018, es.wikipedia.org/wiki/Go.

[16] D. Hof, Robert. *Is Artificial Intelligence Finally Coming into Its Own?*. MIT Technology Review, 2018, www.technologyreview.com/s/513696/deep-learning/

[17] *Keras: The Python Deep Learning Library*. Keras Documentation, keras.io/.

[18] *Machine Learning Crash Course | Google Developers*. Google, developers.google.com/machine-learning/crash-course/.

[19] Mozur, Paul. *Google's AlphaGo Defeats Chinese Go Master in Win for A.I.*. New York Times, 7 Aug. 2018, www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html

[20] Murnane, Kevin. *What Is Deep Learning And How Is It Useful?*. Forbes Magazine, 5 Apr. 2016, www.forbes.com/sites/kevinmurnane/2016/04/01/what-is-deep-learning-and-how-is-it-useful/#621baa73d547

[21] Nicolas, et al. *Emergence of Locomotion Behaviours in Rich Environments*. 10 July 2017, arxiv.org/abs/1707.02286

[22] *OpenAI*. OpenAI, openai.com/

[23] *OpenAI Blog*. OpenAI Blog, blog.openai.com/.

[24] OpenAI. *Dota 2*. OpenAI Blog, 2017, blog.openai.com/dota-2/.

[25] OpenAI. *The International 2018: Results*, OpenAI Blog, 2018, blog.openai.com/the-international-2018-results/.

[26] OpenAI. *More on Dota 2*. OpenAI Blog, 2017, blog.openai.com/more-on-dota-2/.

[27] OpenAI. *OpenAI Five*. OpenAI Blog, 2018, blog.openai.com/openai-five/.

[28] OpenAI. *A Toolkit for Developing and Comparing Reinforcement Learning Algorithms.* Gym, gym.openai.com/docs/.

[29] OpenAI. *A Toolkit for Developing and Comparing Reinforcement Learning Algorithms.* Gym, gym.openai.com/envs/CartPole-v1/.

[30] OpenAI. *A Toolkit for Developing and Comparing Reinforcement Learning Algorithms.* Gym, gym.openai.com/envs/Taxi-v2/.

[31] *Playing Atari with Deep Reinforcement Learning.* DeepMind, deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/.

[32] *Recognizing Hand-Written Digits.* Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html#sphx-glr-auto-examples-classification-plot-digits-classification-py

[33] *Support Vector Regression (SVR) Using Linear and Non-Linear Kernels.* Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html.

[34] Udacity. *Udacity/Deep-Learning.* github.com/udacity/deep-learning/blob/master/reinforcement/Q-learning-cart.ipynb.

[35] *Vision API - Image Content Analysis | Cloud Vision API | Google Cloud.* Google, cloud.google.com/vision/.

[36] *An Introduction to Machine Learning with Scikit-Learn.* Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/tutorial/basic/tutorial.html.

[37] *Scikit-Learn.* Scikit-Learn 0.19.1 Documentation, scikit-learn.org/stable/

[38] Palanisamy, Praveen. *Hands-On Intelligent Agents With OpenAI Gym.* Packt Publishing Limited, 2018.

[39] Matplotlib: Python Plotting. matplotlib.org/.

[40] NumPy, www.numpy.org/.

[41] *What Is the Difference between Neural Networks and Deep Learning?.* Quora, www.quora.com/What-is-the-difference-between-Neural-Networks-and-Deep-Learning

[42] Walia, Anish Singh. *Types of Optimization Algorithms Used in Neural Networks and Ways to Optimize Gradient Descent.* Towards Data Science, 2017.

[43] Keras Documentation. `https://keras.io/initializers/`

[44] Keras Documentation. `https://keras.io/optimizers/`

[45] Python Software Foundation. `https://www.python.org/`